[1 基本概念 12](#_Toc533154984)

[1.1 注释 12](#_Toc533154985)

[1.1.1 语法 12](#_Toc533154986)

[1.1.2 C风格 13](#_Toc533154987)

[1.1.3 C++风格 13](#_Toc533154988)

[1.1.4 注意 13](#_Toc533154989)

[1.2 ASCII表 13](#_Toc533154990)

[1.3 名称和标识符 13](#_Toc533154991)

[1.3.1 标识符 13](#_Toc533154992)

[1.3.2 名称 15](#_Toc533154993)

[1.3.3 标识符能使用的Unicode字符 15](#_Toc533154994)

[1.4 类型 15](#_Toc533154995)

[1.4.1 分类 15](#_Toc533154996)

[1.4.2 类型命名 17](#_Toc533154997)

[1.4.3 详细类型说明符 18](#_Toc533154998)

[1.4.4 静态类型 18](#_Toc533154999)

[1.4.5 动态类型 18](#_Toc533155000)

[1.4.6 不完整类型 18](#_Toc533155001)

[1.4.7 基本类型 19](#_Toc533155002)

[1.5 对象-作用域-生命周期 22](#_Toc533155003)

[1.5.1 对象 22](#_Toc533155004)

[1.5.2 作用域 28](#_Toc533155005)

[1.5.3 生命周期 35](#_Toc533155006)

[1.6 定义和ODR 37](#_Toc533155007)

[1.6.1 定义 37](#_Toc533155008)

[1.6.2 一个定义规则（ODR） 39](#_Toc533155009)

[1.6.3 ODR-use 40](#_Toc533155010)

[1.7 名称查找 43](#_Toc533155011)

[1.7.1 限定名称查找 43](#_Toc533155012)

[1.7.2 非限定名称查找 44](#_Toc533155013)

[1.8 存储模型和数据竞争 52](#_Toc533155014)

[1.8.1 字节 52](#_Toc533155015)

[1.8.2 存储单元 52](#_Toc533155016)

[1.8.3 线程和数据竞争 53](#_Toc533155017)

[1.8.4 内存顺序 54](#_Toc533155018)

[1.8.5 Forward progress 54](#_Toc533155019)

[1.9 转译阶段 57](#_Toc533155020)

[1.9.1 阶段1 57](#_Toc533155021)

[1.9.2 阶段2 57](#_Toc533155022)

[1.9.3 阶段3 57](#_Toc533155023)

[1.9.4 阶段4 58](#_Toc533155024)

[1.9.5 阶段5 58](#_Toc533155025)

[1.9.6 阶段6 58](#_Toc533155026)

[1.9.7 阶段7 58](#_Toc533155027)

[1.9.8 阶段8 58](#_Toc533155028)

[1.9.9 阶段9 58](#_Toc533155029)

[1.9.10 注意 59](#_Toc533155030)

[1.9.11 参考 59](#_Toc533155031)

[1.10 main() 函数 59](#_Toc533155032)

[1.11 布局 59](#_Toc533155033)

[1.11.1 标准布局类型 59](#_Toc533155034)

[1.12 文字类型（LiteralType） 60](#_Toc533155035)

[1.12.1 要求 61](#_Toc533155036)

[1.12.2 举例 61](#_Toc533155037)

[2 C++关键字 62](#_Toc533155038)

[2.1 各类关键字 62](#_Toc533155039)

[2.1.1 using关键字 62](#_Toc533155040)

[3 预处理器 63](#_Toc533155041)

[3.1 条件预编译 63](#_Toc533155042)

[3.2 文本宏替换 63](#_Toc533155043)

[3.3其它预处理指令 63](#_Toc533155044)

[4 表达式 63](#_Toc533155045)

[4.0 概述 63](#_Toc533155046)

[4.0.1 操作符 64](#_Toc533155047)

[4.0.2 转换 65](#_Toc533155048)

[4.0.3 内存分配 65](#_Toc533155049)

[4.0.4 其它 65](#_Toc533155050)

[4.0.5 基本表达式 65](#_Toc533155051)

[4.0.6 文字 66](#_Toc533155052)

[4.0.7 不计算表达式 66](#_Toc533155053)

[4.0.8 Discarded-value expressions 67](#_Toc533155054)

[4.1 值类别 68](#_Toc533155055)

[4.1.1 基本类型 69](#_Toc533155056)

[4.1.2 混合类型 71](#_Toc533155057)

[4.1.3 特殊类型 72](#_Toc533155058)

[4.1.4 变更历史 73](#_Toc533155059)

[4.2 操作数计算顺序 74](#_Toc533155060)

[4.2.1 提前序列化规则（C++11之后） 74](#_Toc533155061)

[4.2.2 序列点规则（C++11之前） 77](#_Toc533155062)

[4.3 常量表达式 78](#_Toc533155063)

[4.3.1 核心常量表达式 78](#_Toc533155064)

[4.4 操作符 81](#_Toc533155065)

[4.4.1 赋值 81](#_Toc533155066)

[4.4.2 算术 81](#_Toc533155067)

[4.4.3 自增和自减 81](#_Toc533155068)

[4.4.4 逻辑 81](#_Toc533155069)

[4.4.5 比较 81](#_Toc533155070)

[4.4.6 成员访问 81](#_Toc533155071)

[4.4.7 其它操作符（函数调用，逗号，条件表达式） 81](#_Toc533155072)

[4.4.8 sizeof操作符 82](#_Toc533155073)

[4.4.9 alignof(C++11) 83](#_Toc533155074)

[4.4.10 new 83](#_Toc533155075)

[4.4.11 delete 87](#_Toc533155076)

[4.4.12 typeid 89](#_Toc533155077)

[4.5 操作符重载（unedit） 91](#_Toc533155078)

[4.6 默认比较(C++20) （unedit） 91](#_Toc533155079)

[4.7 运算符优先级（unedit） 91](#_Toc533155080)

[4.8 转换（incomplete） 91](#_Toc533155081)

[4.8.1 隐式转换 91](#_Toc533155082)

[4.8.2 显式转换 95](#_Toc533155083)

[4.8.3 用户自定义 95](#_Toc533155084)

[4.8.4 static\_cast 95](#_Toc533155085)

[4.8.5 dynamic\_cast 100](#_Toc533155086)

[4.8.6 const\_cast 100](#_Toc533155087)

[4.8.7 reinterpret\_cast 100](#_Toc533155088)

[4.9 文字常量 100](#_Toc533155089)

[4.9.1 boolean - integer - floating 100](#_Toc533155090)

[4.9.2 character - string 100](#_Toc533155091)

[4.9.3 nullptr(C++11) 100](#_Toc533155092)

[4.9.4 user-defined (C++11) 100](#_Toc533155093)

[5 声明 100](#_Toc533155094)

[5.1 命名空间 103](#_Toc533155095)

[5.1.1语法 103](#_Toc533155096)

[5.1.2解释 104](#_Toc533155097)

[5.2 命名空间别名 117](#_Toc533155098)

[5.3 左值和右值引用（已完成\*\*） 117](#_Toc533155099)

[5.3.1 语法 117](#_Toc533155100)

[5.3.2 左值引用 118](#_Toc533155101)

[5.3.3 右值引用 119](#_Toc533155102)

[5.3.4 转发引用（C++11） 121](#_Toc533155103)

[5.3.5 悬空引用 122](#_Toc533155104)

[5.4 指针和数组 123](#_Toc533155105)

[5.4.1 指针 123](#_Toc533155106)

[5.4.2 数组 123](#_Toc533155107)

[5.5 结构化绑定（C++17后） 123](#_Toc533155108)

[5.5.1 语法解释 123](#_Toc533155109)

[5.5.2 举例说明 123](#_Toc533155110)

[5.6 枚举 124](#_Toc533155111)

[5.6.1 枚举声明 124](#_Toc533155112)

[5.6.2 没有作用域的枚举 124](#_Toc533155113)

[5.6.3 具有作用域的枚举（C++11之后） 124](#_Toc533155114)

[5.6.4 举例说明 126](#_Toc533155115)

[5.7 存储周期和链接 127](#_Toc533155116)

[5.7.1 存储类关键字 127](#_Toc533155117)

[5.7.2 解释 127](#_Toc533155118)

[5.7.3 存储期 128](#_Toc533155119)

[5.7.4 链接属性 128](#_Toc533155120)

[5.7.5 静态局部变量 129](#_Toc533155121)

[5.7.6 注意 130](#_Toc533155122)

[5.7.7 关键字 130](#_Toc533155123)

[5.7.8 举例说明 131](#_Toc533155124)

[5.8 语言链接 131](#_Toc533155125)

[5.8.1 语法 132](#_Toc533155126)

[5.8.2 解释 132](#_Toc533155127)

[5.9 内嵌汇编 134](#_Toc533155128)

[5.10 CV类型限定符–常量表达式 135](#_Toc533155129)

[5.10.1 const和volatile类型限定符 135](#_Toc533155130)

[5.10.2 解释 135](#_Toc533155131)

[5.10.3 常量表达式限定符（C++11之后） 136](#_Toc533155132)

[5.11 decltype-auto(C++11) 137](#_Toc533155133)

[5.11.1 decltype操作符 137](#_Toc533155134)

[5.12 alignas(C++11) 139](#_Toc533155135)

[5.13类型别名 139](#_Toc533155136)

[5.13.1 typedef修饰符 139](#_Toc533155137)

[5.13.2 类型别名声明(C++11) 140](#_Toc533155138)

[5.14 详细类型说明符（未完待续） 142](#_Toc533155139)

[5.15 属性(C++11) 142](#_Toc533155140)

[5.16 static\_assert(C++11) 142](#_Toc533155141)

[6 初始化 142](#_Toc533155142)

[6.1 默认初始化 142](#_Toc533155143)

[6.2 值初始化(C++03) 142](#_Toc533155144)

[6.3 拷贝初始化 142](#_Toc533155145)

[6.4 Direct initialization 142](#_Toc533155146)

[6.5 Aggregate initialization 142](#_Toc533155147)

[6.6 列表初始化(C++11) 142](#_Toc533155148)

[6.7 引用初始化 142](#_Toc533155149)

[6.7.1 语法 142](#_Toc533155150)

[6.7.2 解释 143](#_Toc533155151)

[6.7.3 临时对象的生命周期 145](#_Toc533155152)

[6.7.4 注意 146](#_Toc533155153)

[6.7.5 例子 146](#_Toc533155154)

[6.8 Static non-local initialization 147](#_Toc533155155)

[6.9 zero - constant 147](#_Toc533155156)

[6.10 Dynamic non-local initialization 147](#_Toc533155157)

[6.11 ordered - unordered 147](#_Toc533155158)

[7 函数 147](#_Toc533155159)

[7.1 函数声明 147](#_Toc533155160)

[7.2.1 函数声明 147](#_Toc533155161)

[7.2.2 返回类型推导（未完待续） 148](#_Toc533155162)

[7.2.3 参数列表（未完待续） 149](#_Toc533155163)

[7.2.4 函数定义（未完待续） 149](#_Toc533155164)

[7.2 默认参数 149](#_Toc533155165)

[7.3 可变参数 149](#_Toc533155166)

[7.4 Lambda表达式(C++11) 149](#_Toc533155167)

[7.4.1 闭包类型的成员 151](#_Toc533155168)

[7.4.2 ClosureType::operator ret(\*)(params)() 152](#_Toc533155169)

[7.4.3 Lambda 捕获 153](#_Toc533155170)

[7.4.4 修改捕获变量 154](#_Toc533155171)

[7.4.5 Lambda的参数 154](#_Toc533155172)

[7.5 inline内联函数 155](#_Toc533155173)

[7.6 参数依赖查找（ADL） 155](#_Toc533155174)

[7.7 重载决策 160](#_Toc533155175)

[7.8 操作符重载 171](#_Toc533155176)

[7.8.1语法 171](#_Toc533155177)

[7.8.2 重载操作符 172](#_Toc533155178)

[7.8.3 限制 173](#_Toc533155179)

[7.8.4 规范 173](#_Toc533155180)

[7.8.5 赋值操作符 173](#_Toc533155181)

[7.8.6 函数调用操作符 173](#_Toc533155182)

[7.8.7 自增和自减 173](#_Toc533155183)

[7.8.8二进制算术运算符 173](#_Toc533155184)

[7.8.9关系运算符 173](#_Toc533155185)

[7.8.10数组下标操作符 173](#_Toc533155186)

[7.8.11 位算术运算符 173](#_Toc533155187)

[7.8.12 布尔运算符 173](#_Toc533155188)

[7.8.13 很少重载的运算符 173](#_Toc533155189)

[7.8.14 举例 173](#_Toc533155190)

[7.9 重载集的地址 173](#_Toc533155191)

[8 语句 174](#_Toc533155192)

[8.1 条件语句 174](#_Toc533155193)

[8.2 for - range-for(C++11) 174](#_Toc533155194)

[8.3 while - do-while 174](#_Toc533155195)

[8.4 continue - break - goto - return 174](#_Toc533155196)

[8.5 synchronized and atomic(TM TS) 174](#_Toc533155197)

[9 类 174](#_Toc533155198)

[9.1 类和共同体 174](#_Toc533155199)

[9.1.1 类声明 174](#_Toc533155200)

[9.1.2 前置声明 175](#_Toc533155201)

[9.1.3 成员声明 176](#_Toc533155202)

[9.2 注入类名 177](#_Toc533155203)

[9.2.1 解释 177](#_Toc533155204)

[9.2.2 类模板 177](#_Toc533155205)

[9.2.3 注入类名和构造函数 179](#_Toc533155206)

[9.3 类成员 179](#_Toc533155207)

[9.3.1 非静态数据成员 180](#_Toc533155208)

[9.4 Static members - Nested classes 180](#_Toc533155209)

[9.5 派生类 - using-declaration 180](#_Toc533155210)

[9.6 虚函数和抽象类 180](#_Toc533155211)

[9.7 成员访问-友邻 180](#_Toc533155212)

[9.8 重载 180](#_Toc533155213)

[9.8.1 override修饰符(C++11之后) 180](#_Toc533155214)

[9.8.2 final修饰符(C++11之后) 181](#_Toc533155215)

[9.9 位域-this指针 182](#_Toc533155216)

[9.9.1 位域 182](#_Toc533155217)

[9.9.2 this指针 185](#_Toc533155218)

[9.10 构造函数和成员初始化列表 186](#_Toc533155219)

[9.11 默认构造函数和析构函数 186](#_Toc533155220)

[9.12 拷贝构造函数 - Copy assignment 187](#_Toc533155221)

[9.13 移动构造函数(C++11) - Move assignment(C++11) 187](#_Toc533155222)

[9.14 转换构造函数- explicit specifier 187](#_Toc533155223)

[10 模板 187](#_Toc533155224)

[10.1 模板形参和实参 188](#_Toc533155225)

[10.1.1 模板形参（已完成\*\*） 188](#_Toc533155226)

[10.1.2 模板实参 190](#_Toc533155227)

[10.1.3 举例 192](#_Toc533155228)

[10.2 类模板 193](#_Toc533155229)

[10.2.1 语法 193](#_Toc533155230)

[10.2.2 实例化 193](#_Toc533155231)

[10.3 函数模板 196](#_Toc533155232)

[10.3.1 语法 196](#_Toc533155233)

[10.3.2 实例化 196](#_Toc533155234)

[10.4 类成员模板 196](#_Toc533155235)

[10.5 Variable template(C++14) 197](#_Toc533155236)

[10.6 模板参数推导 197](#_Toc533155237)

[10.7 显式特化 197](#_Toc533155238)

[10.7.1 详细内容 198](#_Toc533155239)

[10.7.2 函数模板的显式特化 198](#_Toc533155240)

[10.7.3 模板特化的成员 198](#_Toc533155241)

[10.8 类模板参数推导(C++17) 198](#_Toc533155242)

[10.9 局部特化 198](#_Toc533155243)

[10.9.1语法： 198](#_Toc533155244)

[10.9.2 实参列表 199](#_Toc533155245)

[10.9.3 名称查找 199](#_Toc533155246)

[10.9.4 局部特化顺序 199](#_Toc533155247)

[10.9.5 局部特化的成员 199](#_Toc533155248)

[10.10 Parameter pack 199](#_Toc533155249)

[10.11 Fold-expressions(C++17) 199](#_Toc533155250)

[10.12 Dependent names - SFINAE 199](#_Toc533155251)

[10.13 约束和概念(C++20)(修改中) 199](#_Toc533155252)

[11 异常 200](#_Toc533155253)

[11.1 抛出异常（throw） 200](#_Toc533155254)

[11.2 try-catch 200](#_Toc533155255)

[11.3 function-try 200](#_Toc533155256)

[11.4 noexcept specifier(C++11) 200](#_Toc533155257)

[11.5 noexcept operator(C++11) 200](#_Toc533155258)

[11.6 Dynamic exception specification(until C++17) 200](#_Toc533155259)

[12 其它 200](#_Toc533155260)

[12.1 C++历史 200](#_Toc533155261)

[12.2 扩展命名空间std 200](#_Toc533155262)

[12.3 未定义行为 200](#_Toc533155263)

[12．4 RAII - Rule of three/five/zero 200](#_Toc533155264)

[12.5 As-if规则 200](#_Toc533155265)

[12.5.1 As-if规则 200](#_Toc533155266)

[12.6 空基优化 202](#_Toc533155267)

[12.7 plmpl 204](#_Toc533155268)

[13 补充 204](#_Toc533155269)

[13.1 C++的RTTI观念和用途 204](#_Toc533155270)

[13.1.1什么是RTTI﹖ 204](#_Toc533155271)

[13.1.2 RTTI可能伴随的副作用 205](#_Toc533155272)

[13.1.3 RTTI的常见的使用场合 206](#_Toc533155273)

[13.1.4 RTTI从那里来﹖ 206](#_Toc533155274)

[13.1.5 程序员自己提供的RTTI 207](#_Toc533155275)

[13.1.6 类库提供RTTI 208](#_Toc533155276)

[13.1.7 C++编译器提供RTTI 209](#_Toc533155277)

[13.1.8 RTTI与虚函数表 210](#_Toc533155278)

# 1 基本概念

本章解释了描述C++编程语言的时候常用的专业术语和概念。

C++程序就是一系列包含声明的文本文件（通常包含头文件和源文件）。经过编译，成为可执行程序，通常都是OS调用它的main函数。

C++程序中的某些单词具有特殊含义，这些词被称为关键字。除此之外，其它的可以用作标识符。在编译时，忽略注释。程序中的某些特殊字符必须用转义序列表示。

C++程序的实体是值，对象，引用，函数，枚举器，类型，类成员，模板，模板特化，命名空间，参数包以及“this”指针。预处理器宏不是C++实体。

实体是通过声明引入的，声明将它们与名称相关联并定义它们的属性。定义使用实体所需属性的声明就是[定义](#_1.6.1_定义)。 程序只能包含任何非内联函数或者变量（[它们使用一个定义规则](#_1.6.2_一个定义规则（ODR）)）的一个定义。

函数的定义包括一系列语句，其中一些包含表达式，用于指定程序要执行的计算。

程序中遇到的名称与使用[名称查找](#_1.7_名称查找)引入它们的声明相关联。 每个名称只在程序部分区域内有效，这就是它的作用范围。一些名称具有链接属性，当它们出现在不同的范围或翻译单元中时，它们引用相同的实体。

C ++中的每个对象，引用，函数，表达式都与一个类型相关联，该类型可能是基本的，复合的或用户自定义的，完整的或不完整的等。

命名对象和对象的命名引用被称为变量。

## 1.1 注释

注释作为一种代码内文档存在。在编译时，它们被编译器有效地忽略;它们仅用于读取源代码的人的参考内容。虽然特定的文档不是C++标准的一部分，但有几个工具可以用不同的文档格式解析注释。

### 1.1.1 语法

/\* 注释内容 \*/ (1)

// 注释内容 \n (2)

* （1）常被称为"C-风格"或多行注释。
* （2）常被称为"C++-风格"或单行注释。

所有的注释都会在转译[阶段3](#_1.10.3_阶段3)时被移除，取而代之的是一个空白字符。

### 1.1.2 C风格

C风格的评论通常用于注释大段文本，但是，它们也可以用来注释单行代码。要插入一个C风格的注释，只需用/\*和\*/来包括文本即可；这将导致注释的内容被编译器忽略。虽然它不是C++标准的一部分，但是/\*\*和\*/经常用于说明文件；这是合法的，因为第2个星号被简单地视为注释的一部分。C风格的注释不能嵌套。

### 1.1.3 C++风格

C++风格的注释通常用于注释单行代码，但是多个C++风格的注释可以放在一起形成多行注释。C++风格的注释告诉编译器忽略//和新行之间的所有内容。

### 1.1.4 注意

因为在预处理器阶段之前删除注释，所以不能使用宏来进行注释，并且没有结束的C样式注释不会从#include的文件中溢出。

除了注释外，用于注释掉源代码的其它方法还有

（1）条件编译

|  |
| --- |
| #if 0  std::cout << "this will not be executed or even compiled\n";  #endif |

和

|  |
| --- |
| if(false) {  std::cout << "this will not be executed\n"  } |

## 1.2 ASCII表

（略）。

## 1.3 名称和标识符

### 1.3.1 标识符

标识符由数字，下划线，小写字母和大写字母，和大部分的Unicode字符等组成的任意序列（详细参见如下）。合法的标识符不能以数字开头，而是以拉丁字母，下划线，或者Unicode编码的非数字字母开始。标识符是大小写敏感的，小写标识符和大写标识符是不同的。

注意：C++语法在形式上要求使用\u或\U转义Unicode字符，但是由于[转译阶段1](#_1.10.1_阶段1)，源代码中的原始unicode字符将呈现给编译器。还要注意，这个特征的支持可能是有限的，例如，[gcc](https://gcc.gnu.org/wiki/FAQ#What_is_the_status_of_adding_the_UTF-8_support_for_identifier_names_in_GCC.3F)。

#### 1.3.1.1 声明

标识符可用于命名对象，引用，函数，枚举器，类型，类成员，命名空间，模板，模板特化，参数包，goto标签和其它实体，但以下情况除外：

* 作为关键字的标识符不能用于其他用途；
* 带有双下划线的标识符保留；
* 以下划线和大写字母开头的标识符被保留；
* 以下划线开头的标识符保留在全局名称空间中。

“保留”在这里的意思是，因为标准库文件内部需要，它们宏定义（#define）或者声明这样的标识符，编译器可以预先定义非标准类型的标识符，并且名称矫直算法可以假定这些标识符中的一些未被使用。如果程序员使用这样的标识符，则行为是未定义的。

另外，#define或#undef和关键字相同的名称是未定义的行为。如果包含了至少一个标准库头文件，#define或#undef和声明在标准库头文件里的名称相同的标识符是未定义的行为。

#### 1.3.1.2 表达式

一个命名变量、函数、概念的特化（C++20之后）、或枚举器的标识符可以被用作表达式中。仅由标识符构成的表达式返回该标识符命名的实体。如果该标识符命名了函数、变量、或数据成员，那么表达式的值类别就是[左值](#_4.1.1.1_左值)，否则就是纯右值（prvalue）（例如，枚举器就是纯右值表达式，概念的特化是布尔型纯右值（C++20之后））。

在非静态成员函数体内，每一个非静态成员的标识符都会隐式地转化成类成员访问表达式，this->member。

#### 1.3.1.3 未限定标识符

除了正确声明的标识符外，以下内容还可以以相同角色用于表达式中：

* 函数表达式里的重载运算符，如***operator*+** 或 ***operator* new**；
* 用户定义的转换函数名称，诸如***operator bool***；
* 用户定义的文字操作符名称，诸如***operator "" \_km***；
* 带有参数列表的模板名称，诸如***MyTemplate<int>***；
* 字符“~”紧跟类名称，诸如***~MyClass***；
* 字符“~”紧跟[decltype](#_5.11_decltype-auto)说明符，诸如***~decltype(str).***

它们和标识符一起被称为非限定标识符表达式（unqualified id-expressions）。

#### 1.3.1.4 限定标识符

一个限定标识符表达式是在非限定标识符的前面加上作用域解析符：：，通过作用域解析符把其与枚举序列，类或命名空间名称或decltype表达式（C++11）分开。例如，表达式std：：string：：npos在命名空间std的类string中命名了静态成员npos的表达式。表达式：：tolower在全局命名空间中命名了函数tolower。表达式：：std：：cout在命名空间std中命名了全局变量cout。表达式boost：：signals2：：connection在命名空间boost声明的signals2命名空间中命名了类型connection。

根据需要，关键字模板可能会出现在限定标识符中以区分依赖的模板名称。

对于限定标识符的名称查找的详细信息，请参阅[限定查找](#_1.7_名称查找)。

### 1.3.2 名称

名称包含以下几种情况，每一种情况必与一个实体或者标签相关联：

* 标识符；
* 函数表达式里的重载运算符的名称 (例如operator+, operator new)；
* 用户自定义的转换函数 (operator bool)；
* 用户定义的文字操作符名称(operator "" \_km)；
* 后跟参数列表的模板名称 (MyTemplate<int>)。

程序中每一个能够表示实体（entity）的名称，都是通过声明引入程序的。表示标签的每一个名称都是通过goto语句或者标签语句引入的。在多个翻译单元中使用的名称可以指相同或不同的实体，这取决于[链接](#_5.7_存储周期和链接)。

当编译器在程序中遇到一个未知的名字时，除了模板声明和定义中的依赖名称之外，它将该未知的名称与通过[名称查找](#_1.7_名称查找)引入的声明关联起来。（对于这些名称，由编译器确定它们是类型，模板，还是其它实体，这可能需要严格的消除歧义）。

### 1.3.3 标识符能使用的Unicode字符

下面的Unicode字符可以在标识符中使用：

|  |  |  |
| --- | --- | --- |
| 编码 | 描述 | 字符 |
| U+00A8 | DIARESIS | ¨ |
| U+00AA | FEMININE ORDINAL INDICATOR | ª |
| U+00AD | SOFT HYPHEN |  |
| U+00AF | MACRON | ¯ |
| U+00B2-U+00B5 | SUPERSCRIPT TWO - MICRO SIGN | ²³´µ |

（未完）

## 1.4 类型

对象，引用，[函数](#_7_函数)包括[函数模板特化](#_10.6_Explicit_specialization)和表达式都有类型属性，这些限制实体所允许的操作，并为其它通用的位序列提供语义。

### 1.4.1 分类

C++类型由以下组成：

* 基本类型(查看std：：is\_fundamental)：
* void类型(查看std：：is\_void)；
* std：：nullptr\_t类型(C++11之后) (查看std：：is\_null\_pointer)；
* 算术类型(查看std：：is\_arithmetic)：
* 浮点类型(float, double, long double) (查看std：：is\_floating\_point)；
* 整数类型(查看std：：is\_integral)：
* bool型；
* 字符类型：
* 窄字符类型(char, signed char, unsigned char)；
* 宽字符类型(char16\_t, char32\_t, wchar\_t)；
* 有符号整形(short int, int, long int, long long int)；
* 无符号整形(unsigned short int, unsigned int, unsigned long int, unsigned long long int)；
* 复合类型(查看std：：is\_compound)：
* 引用类型(查看std：：is\_reference)：
  + [左值引用类型](#_5.3.2_左值引用)(查看std：：is\_lvalue\_reference)：
    - 对象左值引用类型；
    - 函数左值引用类型；
* [右值引用类型](#_5.3.3_右值引用)(查看std：：is\_rvalue\_reference)：
* 对象右值引用类型；
* 函数右值引用类型；
* 指针类型(查看std：：is\_pointer)：
* 对象指针类型；
* 函数指针类型；
* 成员指针类型(查看std：：is\_member\_pointer)：
* 数据成员指针类型(查看std：：is\_member\_object\_pointer)；
* 函数成员指针类型(查看std：：is\_member\_function\_pointer)；
* [数组](#_5.4.2_数组)类型(查看std：：is\_array)；
* [函数](#_7_函数)类型(查看std：：is\_function)；
* [枚举](#_5.6_枚举)类型(查看std：：is\_enum)；
* [类](#_9_类)类型：
* 非联合类型(查看std：：is\_class)；
* 联合类型(查看std：：is\_union)。

对于除引用和函数以外的每种类型，类型系统都支持该类型的另外三个[cv限定版本](#_5.10_CV类型限定符–常量表达式)（const，volatile和const volatile）。

类型根据属性，分为不同的类别：

* 对象类型（可能是cv限定的）是非函数类型，非引用类型和非void类型（另请参阅std：：is\_object）
* 标量类型（可能是cv限定的）是算术，指针，成员指针，枚举和std::nullptr\_t类型（另请参阅std::is\_scalar）；
* 普通类型（参见std::is\_trivial），POD类型（另请参阅std::is\_pod），文字类型（另请参阅std::is\_literal\_type）以及类型traits库中列出的其他类别或命名类型要求。

### 1.4.2 类型命名

可以通过下面的方法声明名称从而引用一个类型：

* [class](#_9.1.1_类声明)声明；
* [enum](#_5.6_枚举)声明；
* [typedef](#_5.13.1_typedef说明符)声明；
* type alias声明。

在C++程序中，经常需要引用没有名称的类型，其语法称为type-id。命名类型为T的type-id语法正好是T类型变量或函数声明的语法，省略了标识符，但声明语法的decl-specifier-seq被限制为type-specifier- seq，并且只要type-id出现在非模板类型别名声明的右侧时就可以定义新类型。

|  |
| --- |
| int\* p; // 指向整数的指针声明  static\_cast<int\*>(p); // type-id是"int\*"  int a[3]; // 三个int型元素的数组声明  new int[3]; // type-id是"int[3]" (被new-type-id调用)    int (\*(\*x[2])())[3]; // 2个指向函数的指针元素的数组  // 返回指向3个int元素的数组指针  new (int (\*(\*[2])())[3]); // type-id是"int (\*(\*[2])())[3]"    void f(int); // 函数的声明，形参为int，返回值类型为void  std::function<void(int)> x = f; // 类型模板参数是type-id "void(int)"  std::function<auto(int) -> void> y = f; // same    std::vector<int> v; // int型vector的声明  sizeof(std::vector<int>); // type-id是"std::vector<int>"    struct { int x; } b; // 创建一个新类型且声明一个新类型的对象b  sizeof(struct{ int x; }); // 错误:无法在sizeof表达式中定义新类型  using t = struct { int x; }; // 创建新类型且声明t为新类型的别名  sizeof(static int); // 错误: 存储类说明符不能作为type-specifier-seq的一部分  std::function<inline void(int)> f; // 错误: 函数说明符也不能作为type-specifier-seq的一部分 |

剥除了名称的声明语法中的声明符被称为抽象声明符。

Type-id可用于以下情况：

* 在cast表达式中指定目标类型；
* 作为[sizeof](#_4.4.6_sizeof), [alignof](#_4.4.7_alignof(C++11)), alignas, new,和typeid的参数；
* 在[类型别名](#_5.13.2_类型别名声明(C++11))声明的右侧；
* 作为函数声明的尾随返回类型；
* 作为模板类型参数的默认参数；
* 作为模板类型参数的模板参数；
* 在[动态异常规范](#_11.6_Dynamic_exception)中。

Type-id在下列情况下能够修改使用：

* 函数的参数列表里(当参数名称被省略时), type-id使用decl-specifier-seq代替type-specifier-seq (特殊地，一些存储类说明符是允许的)；
* 用户自定义的转换函数的名称里，抽象声明符不能包含函数或数组操作符。

### 1.4.3 详细类型说明符

Elaborated type specifiers may be used to refer to a previously-declared class name (class, struct, or union) or to a previously-declared enum name even if the name was hidden by a non-type declaration. They may also be used to declare new class names.

详细类型说明符可用于引用先前声明的类名称（类，结构或联合）或先前声明的枚举名称，即使名称被非类型声明隐藏。它们也可能用于声明新的类名。

详细请看[详细类型说明符](#_5.14_详细类型说明符)。

### 1.4.4 静态类型

从程序的编译时分析得出的表达式的类型被称为表达式的静态类型。 程序执行时，静态类型不会改变。

### 1.4.5 动态类型

如果某一[泛左值表达式](#_4.1.2.1_泛左值（glvalue）)引用多态对象，则其最底层派生对象的类型称为动态类型。

|  |
| --- |
| struct B { virtual ~B() {} }; // 多态类型  struct D: B {}; // 多态类型  D d; // 最底层的对象  B\* ptr = &d;  // the static type of (\*ptr) is B  // the dynamic type of (\*ptr) is D |

对于纯右值表达式，动态类型总是和静态类型一样的。

### 1.4.6 不完整类型

下面的类型属于不完整类型：

* void（也许是cv限定的）；
* 已声明的class类型（例如，前置声明），但是没有被定义；
* [未知范围的数组](#_5.4.2.5_无界数组)；
* 具有不完整类型元素的数组；
* 枚举类型，从声明处开始到它潜在的类型被确定。

下面的内容要求class T必须是完整类型：

* definition or function call to a function with return type T or argument type T；
* 类型T的对象定义；
* 类型T的非静态类数据成员的声明；
* new一个类型为T的对象的表达式或者new一个元素类型为T的数组；
* 应用到类型为T的泛左值的左值向右值的转换；
* 类型T的隐式或显式转换；
* 类型T\*或T&的标准转换，dynamic\_cast，static\_cast，例外就是，从null指针常量或void指针开始转换时；
* 类成员访问运算符应用于类型T的表达式；
* Typeid，sizeof，或alignof应用到类型T；
* 算术操作符应用到T指针；
* 拥有基类T的类定义；
* 赋值给类型为T的左值；
* 对类型T，T&，或T\*的异常捕获（catch子句）。

(一般来说，T的大小和布局必须是已知的。)

如果这些情况中的任何一个出现在转译单元里，类型的定义必须出现在相同的转译单元里。

### 1.4.7 基本类型

（另请参阅类型系统概述的类型以及由C++库提供的与类型相关的实用程序的列表）

#### 1.4.7.1 void

void - type with an empty set of values. It is an incomplete type that cannot be completed (consequently, objects of type void are disallowed). There are no arrays of void, nor references to void. However, pointers to void and functions returning type void (procedures in other languages) are permitted.

void –空集类型。这是一个不完整的类型（因此，void类型的对象是不允许的）。没有void的数组，也没有引用void。 但是，指向void和函数的指针返回类型void（其他语言的过程）是允许的。

std：：nullptr\_t

Defined in header <cstddef>

typedef decltype(nullptr) nullptr\_t； (since C++11)

std：：nullptr\_t is the type of the null pointer literal, nullptr. It is a distinct type that is not itself a pointer type or a pointer to member type.

#### 1.4.7.2 布尔

bool - type, capable of holding one of the two values： true or false. The value of sizeof(bool) is implementation defined and might differ from 1.

#### 1.4.7.3 字符

signed char - type for signed character representation.

unsigned char - type for unsigned character representation. Also used to inspect object representations (raw memory).

char - type for character representation which can be most efficiently processed on the target system (has the same representation and alignment as either signed char or unsigned char, but is always a distinct type). Multibyte characters strings use this type to represent code units. The character types are large enough to represent any UTF-8 code unit (since C++14). The signedness of char depends on the compiler and the target platform： the defaults for ARM and PowerPC are typically unsigned, the defaults for x86 and x64 are typically signed.

wchar\_t - type for wide character representation (see wide strings). Required to be large enough to represent any supported character code point (32 bits on systems that support Unicode. A notable exception is Windows, where wchar\_t is 16 bits and holds UTF-16 code units) It has the same size, signedness, and alignment as one of the integer types, but is a distinct type.

char16\_t - type for UTF-16 character representation, required to be large enough to represent any UTF-16 code unit (16 bits). It has the same size, signedness, and alignment as std：：uint\_least16\_t, but is a distinct type. (since C++11)

char32\_t - type for UTF-32 character representation, required to be large enough to represent any UTF-32 code unit (32 bits). It has the same size, signedness, and alignment as std：：uint\_least32\_t, but is a distinct type. (since C++11)

#### 1.4.7.3 整数

int - 基本整数类型。如果使用下面列出的任何修饰符，则可以省略关键字int。如果不存在长度修饰符，则保证其宽度至少为16位。但是，在32/64位系统中，几乎只能保证至少有32位的宽度（见下文）。

**1修饰符**

修改整数类型。 可以按任何顺序混合使用。 每个组中只有一个可以出现在类型名称中。

**2符号**

signed -目标类型将是有符号表示（如果省略，则这是默认值）

unsigned -目标类型将是无符号表示

**3 大小**

short -目标类型将针对空间进行优化并且将具有至少16位的宽度。

long -目标类型的宽度至少为32位。

long long -目标类型的宽度至少为64位。（C++11后）

**注意：与所有类型说明符一样，任何顺序都是允许的：unsigned long long int和long int unsigned long命名了相同的类型**。

**4 属性**

下表总结了所有可用的整数类型及其属性：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类型修饰符 | 等价类型 | 数据模型中的位宽 | | | | |
| C++ 标准 | LP32 | ILP32 | LLP64 | LP64 |
| short | short int | 至少16 | 16 | 16 | 16 | 16 |
| short int |
| signed short |
| signed short int |
| unsigned short | unsigned short int |
| unsigned short int |
| int | int | 至少16 | 16 | 32 | 32 | 32 |
| signed |
| signed int |
| unsigned | unsigned int |
| unsigned int |
| long | long int | 至少32 | 32 | 32 | 32 | 64 |
| long int |
| signed long |
| signed long int |
| unsigned long | unsigned long int |
| unsigned long int |
| long long | long long int(C++11) | 至少64 | 64 | 64 | 64 | 64 |
| long long int |
| signed long long |
| signed long long int |
| unsigned long long | unsigned long long int(C++11) |
| unsigned long long int |

除了最小的位数外，C++标准还必须保证：

|  |
| --- |
| 1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long) |

**注意1：这允许极端情况下字节大小为64位，所有类型（包括char）都是64位宽，并且sizeof对于每种类型都返回1**。

**注意2：对于有符号和无符号整数类型，整数算术的定义是不同的。 请参阅**[**算术运算符**](#_4.4.2_算术)**，特别是**[**整数溢出**](#_4.4.2.3_溢出)。

#### 1.4.7.4 数据模型

每种实现对基本类型大小的选择统称为数据模型。 四个数据模型被广泛接受：

32位系统：

* LP32 或2/4/4 (int是16位，long或者指针是32位)
* Win16 API
* ILP32 or 4/4/4（int，long或者指针都是32位）；
* Win32 API
* Unix 和类Unix系统（Linux，Mac OS X）

64位系统：

* LLP64 or 4/4/8 (int and long are 32-bit, pointer is 64-bit)
* Win64 API
* LP64 or 4/8/8 (int is 32-bit, long and pointer are 64-bit)
* Unix and Unix-like systems (Linux, Mac OS X)

其他型号非常少见。例如，ILP64（8/8/8：int，long和指针是64位）仅出现在一些早期的64位Unix系统（例如Cray上的Unicos）中。

#### 1.4.7.5 浮点数

* 浮点 - 单精度浮点类型。通常IEEE-754 32位浮点型
* 双精度浮点型。通常IEEE-754 64位浮点类型
* 长双精度浮点型。扩展精度浮点型。不一定映射到IEEE-754规定的类型 通常x86和x86-64架构上的80位x87浮点类型。

**属性**

浮点数支持特殊数值：

* 无穷大（正，负）参见INFINITY:
* 负零，-0.0。它与正零相等，但在一些算术运算中是有意义的，例如，1.0 / 0.0 == INFINITY，但是1.0 / -0.0 == -INFINITY），并且对于一些数学函数，例如sqrt(std::complex)；
* 非数，NaN，这与任何东西（包括它本身）都不相等。多位模式表示NaN，参见std :: nan，NAN。

## 1.5 对象-作用域-生命周期

### 1.5.1 对象

对象就是C++程序创建，销毁，引用，访问和控制的对象。

C++中的对象就是一块存储区域，具有：

* 大小（可以用sizeof确定大小）；
* 对齐要求(可以用alignof来确定)；
* [存储周期](#_5.7_存储周期和链接) (automatic, static, dynamic, thread-local)；
* 生命周期 (由存储周期和临时类型限制)；
* 类型；
* 值 (这可能是不确定的，例如，被默认初始化的非类类型)；
* 名称（可选）。

下面的实体不是对象：值，引用，函数，枚举器，类型，非静态类成员，位域，模板，类或函数模板特化，命名空间，参数包以及this指针。

变量是一个对象或声明引入的非静态数据成员的引用。

Objects are created by definitions, new-expressions, throw-expressions, when changing the active member of a union, and where temporary objects are required.

对象由定义，new，throw，更改union的活动成员以及需要临时对象的位置创建。

#### 1.5.1.1 对象存储和值表示

对于类型为T的对象，对象表示形式是从与T对象相同的地址开始的unsigned char类型的大小为sizeof（T）的序列（或者等价于std::byte）。

对象的值表示是保存其类型T的值的位集合。

对于浅拷贝类型，值表示是对象表示的一部分，这意味着复制对象在内存中占用的字节足以生成具有相同值的另一个对象（除非该值是其类型的陷阱表示并且将其加载到CPU中会引发硬件异常，例如SNaN（“signalling not-a-number”）浮点值或NaT（“not-a-thing”）整数）。

The reverse is not necessarily true： two objects of TriviallyCopyable type with different object representations may represent the same value. For example, multiple floating-point bit patterns represent the same special value NaN. More commonly, some bits of the object representation may not participate in the value representation at all； such bits may be padding introduced to satisfy alignment requirements, bit field sizes, etc.

反过来却不一定正确：具有不同对象表示的浅拷贝类型的两个对象可以表示相同的值。例如，多个浮点位模式表示相同的特殊值NaN。 更一般地，对象表示的一些比特位根本不参与值表示;这些比特位可以是引入的填充以满足对齐要求，位域大小等。

|  |
| --- |
| #include <cassert>  struct S {  char c; // 1 个字节  // 3 个字节填充  float f; // 4 字节值  bool operator==(const S& arg) const { // 基于值的相等  return c == arg.c && f == arg.f;  }  };  assert(sizeof(S) == 8);  S s1 = {'a', 3.14};  S s2 = s1;  reinterpret\_cast<char\*>(&s1)[2] = 'b'; // 改变第2个字节  assert(s1 == s2); // 值没有改变 |

对于char，signed char和unsigned char类型的对象（除非它们是过大的位域），对象存储的每一位都需要参与值表示，并且每个可能的位组合表示一个不同的值（无填充，陷阱位或允许多重表示）。

#### 1.5.1.2 子对象

一个对象可以包含其它对象，称之为子对象。包括：

* 成员对象
* 基类子对象
* 数组元素

不是另一个对象的子对象的对象称为完整对象。

Complete objects, member objects, and array elements are also known as most derived objects, to distinguish them from base class subobjects. The size of a most derived object that is not a bit field is required to be non-zero (the size of a base class subobject may be zero： see empty base optimization).

为了与基类子对象区分开来，完整对象、成员对象和数组元素也被称为最底层派生对象。最底层派生对象不是一个位域，且要求非零，（基类子对象的大小可能为零：请参阅空基优化）。

Any two objects with overlapping lifetimes (that are not bit fields) are guaranteed to have different addresses unless one of them is a subobject of another or provides storage for another, or if they are subobjects of different type within the same complete object, and one of them is a zero-size base.

static const char c1 = 'x';

static const char c2 = 'x';

assert(&c1 != &c2); // 相同的值，不同的地址

#### 1.5.1.3 多态对象

声明或继承至少一个虚函数的类对象是多态对象。在每个多态对象中，实现存储附加信息（在每个已有实现中，它是一个指针，除非被优化），虚拟函数调用和RTTI特性（dynamic\_cast和typeid）使用该信息来确定在运行时创建对象的类型，无论其使用的表达式如何。

对于非多态对象，值的解释由使用对象的表达式决定，并在编译时决定。

|  |
| --- |
| #include <iostream>  #include <typeinfo>  struct Base1 {  // 多态类型：声明一个virtual成员  virtual ~Base1() {}  };  struct Derived1 : Base1 {  // 多态类型：声明一个virtual成员  };  struct Base2 {  // 非多态类型  };  struct Derived2 : Base2 {  // 非多态类型  };    int main()  {  Derived1 obj1; // 使用类型Derived1创建object1  Derived2 obj2; // 使用类型Derived2创建object2    Base1& b1 = obj1; // b1 引用对象obj1  Base2& b2 = obj2; // b2 引用对象obj2    std::cout << "Expression type of b1: " << typeid(decltype(b1)).name() << ' '  << "Expression type of b2: " << typeid(decltype(b2)).name() << '\n'  << "Object type of b1: " << typeid(b1).name() << ' '  << "Object type of b2: " << typeid(b2).name() << '\n'  << "size of b1: " << sizeof b1 << ' '  << "size of b2: " << sizeof b2 << '\n';  } |

输出：

|  |
| --- |
| Expression type of b1: Base1 Expression type of b2: Base2  Object type of b1: Derived1 Object type of b2: Base2  size of b1: 8 size of b2: 1 |

#include <iostream>

#include <typeinfo>

struct Base1 {

// polymorphic type: declares a virtual member

virtual ~Base1() {}

};

struct Derived1 : Base1 {

// polymorphic type: inherits a virtual member

};

struct Base2 {

// non-polymorphic type

};

struct Derived2 : Base2 {

// non-polymorphic type

};

int main()

{

Derived1 obj1; // object1 created with type Derived1

Derived2 obj2; // object2 created with type Derived2

Base1& b1 = obj1; // b1 refers to the object obj1

Base2& b2 = obj2; // b2 refers to the object obj2

std::cout << "Expression type of b1: " << typeid(decltype(b1)).name() << ' '

<< "Expression type of b2: " << typeid(decltype(b2)).name() << '\n'

<< "Object type of b1: " << typeid(b1).name() << ' '

<< "Object type of b2: " << typeid(b2).name() << '\n'

<< "size of b1: " << sizeof b1 << ' '

<< "size of b2: " << sizeof b2 << '\n';

}

Output：

Expression type of b1: Base1 Expression type of b2: Base2

Object type of b1: Derived1 Object type of b2: Base2

size of b1: 8 size of b2: 1

#### 1.5.1.3 Strict aliasing

Accessing an object using an expression of a type other than the type with which it was created is undefined behavior in many cases, see reinterpret\_cast for the list of exceptions and examples.

#### 1.5.1.4 Alignment

Every object type has the property called alignment requirement, which is an integer value (of type std：：size\_t, always a power of 2) representing the number of bytes between successive addresses at which objects of this type can be allocated. The alignment requirement of a type can be queried with alignof or std：：alignment\_of. The pointer alignment function std：：align can be used to obtain a suitably-aligned pointer within some buffer, and std：：aligned\_storage can be used to obtain suitably-aligned storage.

Each object type imposes its alignment requirement on every object of that type； stricter alignment (with larger alignment requirement) can be requested using alignas.

In order to satisfy alignment requirements of all non-static members of a class, padding may be inserted after some of its members.

#include <iostream>

// objects of type S can be allocated at any address

// because both S.a and S.b can be allocated at any address

struct S {

char a; // size: 1, alignment: 1

char b; // size: 1, alignment: 1

}; // size: 2, alignment: 1

// objects of type X must be allocated at 4-byte boundaries

// because X.n must be allocated at 4-byte boundaries

// because int's alignment requirement is (usually) 4

struct X {

int n; // size: 4, alignment: 4

char c; // size: 1, alignment: 1

// three bytes padding

}; // size: 8, alignment: 4

int main()

{

std::cout << "sizeof(S) = " << sizeof(S)

<< " alignof(S) = " << alignof(S) << '\n';

std::cout << "sizeof(X) = " << sizeof(X)

<< " alignof(X) = " << alignof(X) << '\n';

}

Output：

sizeof(S) = 2 alignof(S) = 1

sizeof(X) = 8 alignof(X) = 4

The weakest alignment (the smallest alignment requirement) is the alignment of char, signed char, and unsigned char, which equals 1； the largest fundamental alignment of any type is the alignment of std：：max\_align\_t. If a type's alignment is made stricter (larger) than std：：max\_align\_t using alignas, it is known as a type with extended alignment requirement. A type whose alignment is extended or a class type whose non-static data member has extended alignment is an over-aligned type. It is implementation-defined if new-expression, std：：allocator：：allocate, and std：：get\_temporary\_buffer support over-aligned types. Allocators instantiated with over-aligned types are allowed to fail to instantiate at compile time, to throw std：：bad\_alloc at runtime, to silently ignore unsupported alignment requirement, or to handle them correctly.

### 1.5.2 作用域

在C++程序中出现的每个名称仅在源代码的一些可能不连续的部分（称为其作用域）中有效。

在范围内，可以使用[非限定名称查找](#_1.7_名称查找)将名称与其声明相关联。

#### 1.5.2.1 块作用域

由一个块（复合语句）中的声明引入的变量的潜在范围从声明开始并在块的结尾处结束。实际作用域与潜在作用域相同，除非存在具有相同名称的声明的嵌套块（在这种情况下，嵌套声明的整个潜在范围将从外部声明的范围中排除）

|  |
| --- |
| int main()  {  int a = 0; // 第一个'a'的作用域开始  ++a; // 名称'a'在作用域内，引用第一个'a'  {  int a = 1; // 第二个'a'的作用域开始  // 第一个'a'的作用域被中断  a = 42; // 名称'a'在作用域内，引用第二个'a'  } // 块结束了，第二个'a'的作用域结束  // 第一个'a'的作用域继续  } // 块结束，第一个'a'的作用域结束  int b = a; // 错误：名称'a'不在作用域内 |

在异常处理程序中声明的名称的潜在作用域从声明开始，在异常处理程序结束时结束，并且不在另一个异常处理程序或封闭块中的作用域中。

|  |
| --- |
| try {  f();  } catch(const std::runtime\_error& re) { // re作用域开始  int n = 1; // n作用域开始  std::cout << re.what(); // re在作用域内  } // re，n的作用域都结束  catch(std::exception& e) {  std::cout << re.what(); // 错误：re不在作用域内  ++n; // 错误：n不在作用域内  } |

在for循环的init语句中，在for循环的条件中，在for循环的范围声明中，在if语句或switch语句的init语句中声明的名称的潜在范围（C++17后）中，在if语句的条件下，while循环或switch语句从声明点开始，并在控制语句结束时结束。

|  |
| --- |
| Base\* bp = new Derived;  if(Derived\* dp = dynamic\_cast<Derived\*>(bp))  {  dp->f(); // dp在作用域内  } // dp作用域结束    for(int n = 0; // n的作用域开始  n < 10; // n在作用域内  ++n) // n在作用域内  {  std::cout << n << ' '; // n在作用域内  } // n作用域结束 |

#### 1.5.2.2 函数参数作用域

The potential scope of a function parameter (including parameters of a lambda expression) or of a function-local predefined variable begins at its point of declaration.

If the nearest enclosing function declarator is not the declarator of a function definition, its potential scope ends at the end of that function declarator.

Otherwise, its potential scope ends at the end of the last exception handler of the function-try-block, or at the end of the function body if a function try block was not used.

函数参数（包括lambda表达式的参数）或函数本地预定义变量的潜在范围从其声明点开始。

如果最近的封闭函数声明符不是函数定义的声明符，则其潜在作用域将在该函数声明符的末尾结束。

否则，如果未使用函数try块，则其潜在作用域将终止于function-try-block的最后一个异常处理程序的末尾，或者在函数体的末尾。

|  |
| --- |
| const int n = 3;    int f1(int n, // scope of global 'n' interrupted,  // scope of the parameter 'n' begins  int y = n); // error: default argument references a parameter    int (\*(\*f2)(int n))[n]; // OK: the scope of the function parameter 'n'  // ends at the end of its function declarator  // in the array declarator, global n is in scope  // (this declares a pointer to function returning a pointer to an array of 3 int    // by contrast  auto (\*f3)(int n)->int (\*)[n]; // error: parameter 'n' as array bound      int f(int n = 2) // scope of 'n' begins  try // function try block  { // the body of the function begins  ++n; // 'n' is in scope and refers to the function parameter  {  int n = 2; // scope of the local variable 'n' begins  // scope of function parameter 'n' interrupted  ++n; // 'n' refers to the local variable in this block  } // scope of the local variable 'n' ends  // scope of function parameter 'n' resumes  } catch(...) {  ++n; // n is in scope and refers to the function parameter  throw;  } // last exception handler ends, scope of function parameter 'n' ends  int a = n; // Error: name 'n' is not in scope |

#### 1.5.2.3 函数作用域

A label (and only a label) declared inside a function is in scope everywhere in that function, in all nested blocks, before and after its own declaration.

在函数中声明的标签（只有标签）在该函数中的所有范围内，在所有嵌套块中，在它自己的声明之前和之后。

|  |
| --- |
| void f()  {  {  goto label; // label in scope even though declared later  label:;  }  goto label; // label ignores block scope  }    void g()  {  goto label; // error: label not in scope in g()  } |

#### 1.5.2.4 命名空间作用域

The potential scope of any entity declared in a namespace begins at the declaration and consists of the concatenation of all namespace definitions for the same namespace name that follow, plus, for any using-directive that introduced this name or its entire namespace into another scope, the rest of that scope.

The top-level scope of a translation unit ("file scope" or "global scope") is also a namespace and is properly called "global namespace scope". The potential scope of any entity declared in the global namespace scope begins at the declaration and continues to the end of the translation unit.

The scope of an entity declared in an unnamed namespace or in an inline namespace includes the enclosing namespace;

在一个名称空间中声明的任何实体的潜在作用域从声明开始，包含相同名称空间名称的所有名称空间定义的连接，以及对于将该名称或其全部名称空间引入到另一个作用域的任何using-directive， 该范围的其余部分。

翻译单元（“文件范围”或“全局范围”）的顶级范围也是一个名称空间，并且正确地称为“全局名称空间范围”。 在全局命名空间范围内声明的任何实体的潜在范围从声明开始并一直延续到翻译单元的末尾。

在未命名的名称空间或内联命名空间中声明的实体的范围包括封闭的名称空间;

|  |
| --- |
| namespace N { // scope of N begins (as a member of global namespace)  int i; // scope of i begins  int g(int a) { return a; } // scope of g begins  int j(); // scope of j begins  void q(); // scope of q begins  namespace {  int x; // scope of x begins  } // scope of x does not end  inline namespace inl { // scope of inl begins  int y; // scope of y begins  } // scope of y does not end  } // scope of i,g,j,q,inl,x,y interrupted    namespace {  int l=1; // scope of l begins  } // scope of l does not end (it's a member of unnamed namespace)    namespace N { // scope of i,g,j,q,inl,x,y continues  int g(char a) { // overloads N::g(int)  return l+a; // l from unnamed namespace is in scope  }  int i; // error: duplicate definition (i is already in scope)  int j(); // OK: repeat function declaration is allowed  int j() { // OK: definition of the earlier-declared N::j()  return g(i); // calls N::g(int)  }  int q(); // error: q is already in scope with different return type  } // scope of i,g,j,q,inl,x,y interrupted    int main() {  using namespace N; // scope of i,g,j,q,inl,x,y resumes  i = 1; // N::i is in scope  x = 1; // N::(anonymous)::x is in scope  y = 1; // N::inl::y is in scope  inl::y = 2; // N::inl is also in scope  } // scope of i,g,j,q,inl,x,y interrupted |

#### 1.5.2.5 类作用域

The potential scope of a name declared in a class begins at the point of declaration and includes the rest of the class body and all function bodies (even if defined outside the class definition or before the declaration of the name), default arguments, exception specifications, in-class brace-or-equal initializers, and all these things in nested classes, recursively.

在类中声明的名称的潜在范围从声明开始，包括类体的其余部分和所有函数体（即使定义在类定义之外或声明名称之前），默认参数，异常规范 ，类内括号或相等的初始值设定项，以及嵌套类中的所有这些东西，递归地。

|  |
| --- |
| class X {  int f(int a = n) { // X::n is in scope inside default parameter  return a\*n; // X::n is in scope inside function body  }  int g();  int i = n\*2; // X::n is in scope inside initializer    // int x[n]; // Error: n is not in scope in class body  static const int n = 1;  int x[n]; // OK: n is now in scope in class body  };  int X::g() { return n; } // X::n is in scope in out-of-class member function body |

If a name is used in a class body before it is declared, and another declaration for that name is in scope, the program is ill-formed, no diagnostic required.

如果一个名称在声明之前在类体中使用，并且该名称的另一个声明在范围内，则该程序不合格，不需要诊断。

|  |
| --- |
| typedef int c; // ::c  enum { i = 1 }; // ::i  class X {  char v[i]; // Error: at this point, i refers to ::i  // but there is also X::i  int f() {  return sizeof(c); // OK: X::c, not ::c is in scope inside a member function  }  char c; // X::c  enum { i = 2 }; // X::i  };    typedef char\* T;  struct Y {  T a; // error: at this point, T refers to ::T  // but there is also Y::T  typedef long T;  T b;  }; |

Names of any class members can only be used in four contexts:

in its own class scope or in the class scope of a derived class

after the . operator applied to an expression of the type of its class or a class derived from it

after the -> operator applied to an expression of the type of pointer to its class or pointers to a class derived from it

after the :: operator applied to the name of its class or the name of a class derived from it

任何班级成员的姓名只能在四种情况下使用：

* 在它自己的类范围或派生类的类范围中
* 之后 。 运算符应用于其类的类型或从其派生的类的表达式
* ->运算符应用于指向其类的指针类型的表达式或指向从其派生的类的指针后
* ::运算符应用于其类的名称或从其派生的类的名称之后

#### 1.5.2.6 枚举作用域

The name of an enumerator introduced in a scoped enumeration begins at the point of declaration and ends at the end of the enum specifier (in contrast, unscoped enumerators are in scope after the end of the enum specifier)

在范围枚举中引入的枚举器的名称从声明开始，并在枚举说明符的结尾处结束（相反，枚举说明符在枚举说明符结束后处于范围内）

|  |
| --- |
| enum e1\_t { // unscoped enumeration  A,  B = A\*2  }; // scope of A and B does not end    enum class e2\_t { // scoped enumeration  SA,  SB = SA\*2 // SA is in scope  }; // scope of SA and SB ends    e1\_t e1 = B; // OK, B is in scope  // e2\_t e2 = SB; // Error: SB is not in scope  e2\_t e2 = e2\_t::SB; // OK |

#### 1.5.2.7 模板参数作用域

The potential scope of a template parameter name begins immediately at the point of declaration and continues to the end of the smallest template declaration in which it was introduced. In particular, a template parameter can be used in the declarations of subsequent template parameters and in the specifications of base classes, but can't be used in the declarations of the preceding template parameters.

模板参数名称的潜在范围在声明时立即开始，并且一直延续到最小模板声明的末尾。 特别是，模板参数可用于后续模板参数的声明和基类的规范中，但不能用于前述模板参数的声明中。

|  |
| --- |
| template< typename T, // scope of T begins  T\* p, // T can be used for a non-type parameter  class U = T // T can be used for a default type  >  class X : public Array<T> // T can be used in base class name  {  // T can be used inside the body as well  }; // scopes of T and U end, scope of X continues |

The potential scope of the name of the parameter of a template template parameter is the smallest template parameter list in which that name appears

|  |
| --- |
| template< template< // template template parameter  typename Y, // scope of Y begins  typename G = Y // Y is in scope  > // scopes of Y and G end  class T,  // typename U = Y // Error: Y is not in scope  typename U  >  class X  {  }; // scopes of T and U end |

Similar to other nested scopes, the name of a template parameter hides the same name from the outer scope for the duration of its own:

与其他嵌套范围类似，模板参数的名称在其自己的持续时间内与外部范围隐藏相同的名称：

|  |
| --- |
| typedef int N;  template< N X, // non-type parameter of type int  typename N, // scope of this N begins, scope of ::N interrupted  template<N Y> class T // N here is the template parameter, not int  > struct A; |

#### 1.5.2.8 声明点

Scope begins at the point of declaration, which is located as follows:

For variables and other names introduced by simple declarations, the point of declaration is immediately after that name's declarator and before its initializer, if any:

范围从声明的角度开始，位置如下：

对于通过简单声明引入的变量和其他名称，声明点紧接在该名称的声明器之后，并且在其初始化器之前（如果有的话）：

|  |
| --- |
| unsigned char x = 32; // scope of the first 'x' begins  {  unsigned char x = x; // scope of the second 'x' begins before the initializer (= x)  // this does not initialize the second 'x' with the value 32,  // this initializes the second 'x' with its own, indeterminate, value  }  std::function<int(int)> f = [&](int n){return n>1 ? n\*f(n-1) : n;};  // the name of the function 'f' is in scope within the lambda, and can  // be correctly captured by reference, giving a recursive function |

|  |
| --- |
| unsigned char x = 32; // scope of the first 'x' begins  {  unsigned char x = x; // scope of the second 'x' begins before the initializer (= x)  // this does not initialize the second 'x' with the value 32,  // this initializes the second 'x' with its own, indeterminate, value  }  std::function<int(int)> f = [&](int n){return n>1 ? n\*f(n-1) : n;};  // the name of the function 'f' is in scope within the lambda, and can  // be correctly captured by reference, giving a recursive function |

The point of declaration of a class or template is immediately after the identifier that names the class (or the template-id that names the template) appears in its class-head, and is already in scope in the list of the base classes:

类或模板的声明点紧跟在用于命名类的标识符（或者称为模板的模板标识符）出现在其头类中，并且已经在基类列表的范围内：

|  |
| --- |
| // the name 'S' is in scope immediately after it appears,  // so it can be used in the list of base classes  struct S: std::enable\_shared\_from\_this<S>  {  }; |

The point of declaration of an enumeration is immediately after the identifier that names it appears in the enum specifier or opaque enum declaration, whichever is used first:

枚举声明的位置紧跟在名称标识符出现在枚举说明符或不透明枚举声明中的那个标识符之后，无论哪个先使用：

|  |
| --- |
| enum E : int { // E is already in scope  A = sizeof(E)  }; |

The point of declaration of a type alias or alias template is immediately after the type-id to which the alias refers:

类型别名或别名模板的声明点紧跟在别名引用的类型标识之后：

|  |
| --- |
| using T = int; // point of declaration of T is at the semicolon  using T = T; // same as T = int |

The point of declaration of an enumerator is immediately after its definition (not before the initializer as it is for variables):

枚举器的声明点紧随其定义之后（不是在初始化器之前，因为它是变量）：

|  |
| --- |
| const int x = 12;  {  enum { x = x + 1, // point of declaration is at the comma, x is initialized to 13  y = x + 1 // the enumerator x is now in scope, y is initialized to 14  };  } |

The point of declaration for an injected-class-name is immediately following the opening brace of its class (or class template) definition

This section is incomplete

Reason: injected-class-name needs to be defined somewhere and linked here

本节不完整

原因：注入类名需要在某处定义并链接到此处

|  |
| --- |
| template<typename T>  struct Array  // : std::enable\_shared\_from\_this<Array> // Error: the injected class name is not in scope  : std::enable\_shared\_from\_this< Array<T> > //OK: the template-name Array is in scope  { // the injected class name Array is now in scope as if a public member name  Array\* p; // pointer to Array<T>  }; |

This section is incomplete

Reason: remainder of [basic.scope.pdecl]

本节不完整

原因：[basic.scope.pdecl]的剩余部分

### 1.5.3 生命周期

Every [object](http://en.cppreference.com/w/cpp/language/object) and [reference](http://en.cppreference.com/w/cpp/language/reference) has a *lifetime*, which is a runtime property： for any object or reference, there is a point of execution of a program when its lifetime begins, and there is a moment when it ends.

* For any object of class or aggregate types if it, or any of its subobjects, is initialized by anything other than the [trivial default constructor](http://en.cppreference.com/w/cpp/language/default_constructor#Trivial_default_constructor), lifetime begins when initialization ends.
* For any object of class types whose [destructor](http://en.cppreference.com/w/cpp/language/destructor#Trivial_destructor) is not trivial, lifetime ends when the execution of the destructor begins.
* Lifetime of a member of a [union](http://en.cppreference.com/w/cpp/language/union) begins when that member is made active
* For all other objects (class objects initialized by a trivial default constructor, non-class objects, arrays of those, etc.), lifetime begins when the properly-aligned storage for the object is allocated and ends when the storage is deallocated or reused by another object.

Lifetime of an object is equal to or is nested within the lifetime of its storage, see [storage duration](http://en.cppreference.com/w/cpp/language/storage_duration).

Lifetime of a reference is exactly its storage duration.

(until C++14)

The lifetime of a reference begins when its initialization is complete and ends as if it were a scalar object.

(since C++14)

Note： the lifetime of the referred object may end before the end of the lifetime of the reference, which makes dangling references possible.

Lifetimes of member objects and base subobjects begin and end following class initialization order.

#### 1.5.3.1 Temporary object lifetime

#### 1.5.3.2 Storage reuse

#### 1.5.3.3生命周期外的访问

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, the following uses of the glvalue expression that identifies that object are undefined:

在对象的生命周期开始之前，但在对象占用了分配给它的存储空间之后，或者在对象的生命周期结束之后并且在重新使用或释放该对象占用的存储空间之前，以下使用glvalue表达式 标识该对象是未定义的：

1. 左值到右值的转换(例如，调用值传递的函数).
2. 访问非静态数据成员或调用非静态成员函数。
3. 将引用绑定到虚基类子对象。
4. dynamic\_cast或typeid表达式。

The above rules apply to pointers as well (binding a reference to virtual base is replaced by implicit conversion to a pointer to virtual base), with two additional rules:

上面的规则也适用于指针（绑定对虚拟基址的引用被隐式转换为指向虚拟基址的指针所替代），还有两条附加规则：

static\_cast of a pointer to storage without an object is only allowed when casting to (possibly cv-qualified) void\*.

Pointers to storage without an object that were cast to possibly cv-qualified void\* can only be static\_cast to pointers to possibly cv-qualified char, possibly cv-qualified unsigned char, or possibly cv-qualified std::byte.

只有在转换为（可能是cv-qualified）void \*时，才允许指向不带对象的存储的指针的static\_cast。

没有被转换为cv-qualified void \*的对象的存储指针只能是static\_cast指向可能是cv-qualified char的指针，可能是cv-qualified unsigned char，或者可能是cv-qualified std :: byte。

During construction and destruction, other restrictions apply, see virtual function calls during construction and destruction.

在施工和销毁期间，其他限制也适用，请参阅施工和销毁期间的虚拟功能调用。

#### 1.5.3.4 注意

The difference in the end of lifetime rules between non-class objects (end of storage duration) and class objects (reverse order of construction) matters in the following example:

在下面的例子中，非类对象（存储持续时间结束）和类对象（构造的反向顺序）之间的生命周期结束时间差异很重要：

|  |
| --- |
| struct A {  int\* p;  ~A() { std::cout << \*p; } // if n outlives a, prints 123  };  void f() {  A a;  int n = 123; // if n does not outlive a, this is optimized out (dead store)  a.p = &n;  } |

## 1.6 定义和ODR

### 1.6.1 定义

定义就是通过声明完整地定义了一个实体。除了下面的情况之外，每一个声明都是一个定义：

（1）没有函数体的函数声明，例：int f(int);

（2）使用extern[存储类说明符](#_5.7.1_存储类关键字)或带有[语言链接](#_5.7.4_链接属性)说明符（如extern“C”）的任何声明（不带初始值）

|  |
| --- |
| extern const int a; // 声明a，但是没有定义  extern const int b = 1; // 定义b |

（3）在类定义中声明非内联（自C++17之后）[静态数据成员](#_9.4_Static_members)

|  |
| --- |
| struct S {  int n; // 定义 S::n  static int i; // 声明，但是没有定义S::i  inline static int x; // 定义S::x  }; // 定义S  int S::i; // 定义S::i |

（4）（不建议使用）使用constexpr说明符在类中定义的静态数据成员的命名空间作用域声明（C++17后）

|  |
| --- |
| struct S {  static constexpr int x = 42; // 隐式内联，定义S::x  };  constexpr int S::x; // 声明S::x，不是重定义 |

（5）声明类名（通过前置声明或在另一个声明中使用详细类型说明符）

|  |
| --- |
| struct S; // 声明，但是没有定义S  class Y f(class T p); // 声明，但是没有定义Y和T(f和p同理) |

（6）枚举的不透明声明（C++11之后）

|  |
| --- |
| enum Color:int; // 声明，但是没有定义Color |

（7）模板参数的声明

|  |
| --- |
| template<typename T> // 声明，但是没有定义T |

（8）函数声明中的参数声明不是定义

|  |
| --- |
| int f(int x); // 声明，但是没有定义f和x  int f(int x) { // 定义f和x  return x+a;  } |

（9）typedef声明

|  |
| --- |
| typedef S S2; // 声明，但是没有定义S2（S可能是不完整的） |

（10）别名声明（C++11后）

|  |
| --- |
| using S2 = S; // 声明，但是没有定义S2（S可能是不完整的） |

（11）using声明

|  |
| --- |
| using N::d; // 声明，但是没有定义d |

（12）Declaration of a deduction guide（没有定义任何实体）（C++17之后）

（13）[static\_assert](#_5.16_static_assert(C++11))声明（没有定义任何实体）（C++11后）

（14）[属性声明](#_5.15_属性(C++11))（没有定义任何实体）（C++11后）

（15）[空声明](#_5_声明)（没有定义任何实体）

（16）[using-directive](#_4_using声明)（没有定义任何实体）

（17）一个明确的实例化声明（一个“extern模板”）（C++11后）

|  |
| --- |
| extern template f<int, char>; // 声明，但是没有定义f<int, char> |

（18）显式特化的声明不是一个定义

|  |
| --- |
| template<> struct A<int>; // 声明，但是没有定义A<int> |

[asm声明](#_5.9_Inline_assembly)没有定义任何实体，但它被归类为定义。

必要时，编译器可以隐式定义默认构造函数，复制构造函数，移动构造函数，复制赋值运算符，移动赋值运算符和析构函数。

如果任何对象的定义导致一个[不完整类型的对象](#_1.4.6_不完整类型)，则该程序是不规范的。

### 1.6.2 一个定义规则（ODR）

（1）任何编译单元（其中一些可能有多个声明，但只允许一个定义）都只允许变量、函数、class、枚举、概念（C++20后）或模板具有唯一定义。

（2）在整个程序（包括任何标准和用户定义的库）中都需要使用一个定义规则（见下文）的每个非内联函数或变量的唯一一个定义。编译器一般不判断，但违反该规则的程序，其行为未定义。

（3）对于内联函数或内联变量（C++17后），如果其是遵循一个定义规则的话，在每一个编译单元都要有一个定义。

（4）在某种程度上，任何编译单元中出现的类都要求是完整的，需要一个类的唯一定义。

（5）一个程序中可以有多个定义，只要每个定义出现在不同的编译单元中，以下各项：class、枚举、具有外部链接的内联函数、具有外部链接的内联变量（C++17后）、类模板、非静态函数模板、类模板的静态数据成员、类模板的成员函数、局部模板特化、概念（C++20后）等，只要满足以下所有条件：

* 每个定义由相同的令牌序列组成（通常出现在相同的头文件中）
* name lookup from within each definition finds the same entities (after overload-resolution), except that constants with internal or no linkage may refer to different objects as long as they are not ODR-used and have the same values in every definition.
* 除了具有内部链接或不链接的常量可以引用不同的对象（只要它们不是ODR使用的并且在每个定义中具有相同的值）之外，每个定义中的名称查找会找到相同的实体（在重载决策之后）。
* overloaded operators, including conversion, allocation, and deallocation functions refer to the same function from each definition (unless referring to one defined within the definition)
* 包括转换，分配和释放函数在内的重载操作符指的是每个定义中的相同函数（除非引用定义中的定义）
* 语言链接属性相同（例如，头文件不在extern “C”块内）
* the three rules above apply to every default argument used in each definition
* 上面的三条规则适用于每个定义中使用的每个默认参数
* if the definition is for a class with an implicitly-declared constructor, every translation unit where it is odr-used must call the same constructor for the base and members
* 如果定义是针对具有隐式声明的构造函数的类，则每个使用odr-used的翻译单元都必须为基础和成员调用相同的构造函数
* if the definition is for a template, then all these requirements apply to both names at the point of definition and dependent names at the point of instantiation
* 如果定义是针对模板的，那么所有这些要求都适用于定义点处的名称和实例化处的相关名称

如果满足所有这些要求，程序的行为就好像整个程序中只有一个定义一样。否则，行为是不确定的。

Note: in C, there is no program-wide ODR for types, and even extern declarations of the same variable in different translation units may have different types as long as they are compatible. In C++, the source-code tokens used in declarations of the same type must be the same as described above: if one .cpp file defines struct S { int x; }; and the other .cpp file defines struct S { int y; };, the behavior of the program that links them together is undefined. This is usually resolved with unnamed namespaces.

### 1.6.3 ODR-use

非正式地说，如果使用一个对象的地址或者引用该对象，对其值进行读取，它就是使用了唯一定义规则。如果一个引用被使用且它的引用体在编译时未知，那么它使用一个定义规则（ODR）；如果发生函数调用或者它的地址被占用，函数就会使用一个定义规则（ODR）。如果一个对象，一个引用或函数使用了一个定义规则（ODR），它的定义必须存在于程序中的某个地方；如果违反，通常是链接时错误。

|  |
| --- |
| struct S {  static const int x = 0; // 静态数据成员  // a definition outside of class is required if it is odr-used  };  const int& f(const int& r);    int n = b ? (1, S::x) // S::x 在这里没有使用唯一定义规则  : f(S::x); // S::x 使用唯一定义规则，只能有一个定义 |

形式上，

1. a variable x in a potentially-evaluated expression ex is odr-used unless both of the following are true除非以下两个条件都成立，:
   * 对变量x应用左值到右值的转换，产生常量表达式，该表达式不会调用有意义的函数
   * either x is not an object (that is, x is a reference) or, if x is an object, it is one of the potential results of a larger expression e, where that larger expression is either a discarded-value expression or has the lvalue-to-rvalue conversion applied to it
   * 要么x不是一个对象（即x是一个引用），或者如果x是一个对象，它是一个更大的表达式e的潜在结果之一，其中较大的表达式是一个[丢弃值表达式](#_4.0.8_Discarded-value_expressions)或者有左值到右值转换应用于它

|  |
| --- |
| struct S { static const int x = 1; }; // 对S::x应用左值向右值的转换，产生一个const表达式  int f() {  S::x; // discarded-value expression does not odr-use S::x  return S::x; // expression where lvalue-to-rvalue conversion applies does not odr-use S::x  } |

1. 如果[\*this](#_9.9.2_this指针)出现在潜在计算的表达式 is odr-used if this appears as a potentially-evaluated expression（括隐含在非静态成员函数调用表达式中的this指针）
2. A structured binding is odr-used if it appears as a potentially-evaluated expression.(since C++17)
3. \*如果这显示为潜在评估的表达式（包括隐含在非静态成员函数调用表达式中），则这是odr使用的。
4. 如果结构化绑定显示为潜在评估的表达式，则该结构是可用的。（自C ++ 17以来）

In the definitions above, potentially-evaluated means the expression is not an unevaluated operand (or its subexpression), such as the operand of sizeof and a set of potential results of an expression e is a (possibly empty) set of id-expressions that appear within e, combined as follows:

* + If e is an id-expression, the expression e is its only potential result
  + If e is an array subscript expression (e1[e2]) where one of the operands is an array, the potential results of that operand is included in the set(since C++17)
  + If e is a class member access expression (e1.e2 or e1->e2), the potential results of the object expression e1 is included in the set.
  + If e is a pointer-to-member access expression (e1.\*e2 or e1->\*e2) whose second operand is a constant expression, the potential results of the object expression e1 are included in the set
  + If e is an expression in parentheses ((e1)), the potential results of e1 are included in the set
  + If e is a glvalue conditional expression (e1?e2:e3, where e2 and e3 are glvalues), the union of the potential results of e2 and e3 are both included in the set.
  + If e is a comma expression (e1,e2), the potential results of e2 are in the set of potential results
  + Otherwise, the set is empty.

|  |
| --- |
| struct S {  static const int a = 1;  static const int b = 2;  };  int f(bool x) {  return x ? S::a : S::b;  // x is a part of the subexpression "x" (to the left of ?),  // which applies lvalue-to-rvalue conversion, therefore x is not odr-used  // S::a and S::b are lvalues, and carry over as "potential results" to the result  // of the glvalue conditional  // That result is then subject to lvalue-to-rvalue conversion requested  // to copy-initialize the return value, therefore S::a and S::b are not odr-used  } |

1. 如果一个函数满足下面的条件，我们就说其遵循“唯一定义规则”：
   * A function whose name appears as a potentially-evaluated expression (including named function, overloaded operator, user-defined conversion, user-defined placement forms of operator new, non-default initialization) is odr-used if it is selected by overload resolution, except when it is an unqualified pure virtual member function or a pointer-to-member to a pure virtual function (since C++17).
   * 虚成员函数不是纯虚成员函数（需要虚成员函数的地址构建虚成员函数表-vtable）
   * An allocation or deallocation function for a class is odr-used by a new expression appearing in a potentially-evaluated expression
   * A deallocation function for a class is odr-used by a delete expression appearing in a potentially-evaluated expression
   * A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class.
   * A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor
   * An assignment operator in a class T that is a member or base of another class U is odr-used by an implicitly-defined copy-assignment or move-assignment functions of U.
   * A constructor (including default constructors) for a class is odr-used by the initialization that selects it.
   * A destructor for a class is odr-used if it is potentially invoked

In all cases, a constructor selected to copy or move an object is odr-used even if copy elision takes place.

## 1.7 名称查找

名称查找就是把程序中遇到的[名称](#_1.3.2_名称)与引入它的[声明](#_5_声明)结合的过程。

例如，为了编译std：：cout << std：：endl；，编译器执行如下步骤：

* 为名称std进行非限定名称查找，在头文件<iostream>中发现命名空间std的声明
* 对cout进行限定名称查找，在命名空间std中发现变量声明
* 对endl进行限定名称查找，在命名空间std中发现函数模板声明
* 对操作符<<这个名称进行[参数依赖查找](#_7.6_参数依赖查找（ADL）)，会在命名空间std中发现多个函数模板声明。然后，对std::ostream::operator<<进行限定名称查找，会在类std::ostream中发现多个成员函数声明

[Template argument deduction](http://en.cppreference.com/w/cpp/language/function_template) may also apply, and the set of declarations is passed to [overload resolution](http://en.cppreference.com/w/cpp/language/overload_resolution), which selects the declaration that will be used. [Member access](http://en.cppreference.com/w/cpp/language/access) rules, if applicable, are considered only after name lookup and overload resolution.

For all other names (variables, namespaces, classes, etc), name lookup must produce a single declaration in order for the program to compile. Lookup for a name in a scope finds all declarations of that name, with one exception, known as the "struct hack" or "type/non-type hiding"： Within the same scope, some occurrences of a name may refer to a declaration of a class/struct/union/enum that is not a typedef, while all other occurrences of the same name either all refer to the same variable, non-static data member (since C++14), or enumerator, or they all refer to possibly overloaded function or function template names. In this case, there is no error, but the type name is hidden from lookup (the code must use [elaborated type specifier](http://en.cppreference.com/w/cpp/language/elaborated_type_specifier) to access it).

对于函数和函数模板名称，[名称查找](#_1.7_名称查找)可以将多个声明关联到相同的名称上，并且可以从[参数依赖查找](#_7.6_参数依赖查找（ADL）)中获取额外的声明。也可以应用[模板参数推导](#_10.5_Template_argument)，并将该组声明传递给[重载决策](#_7.7_Overload_resolution)，该决策选择将使用的声明。成员访问规则（如果适用）仅在名称查找和重载解析后才被考虑。

对于所有其他名称（变量，命名空间，类等），名称查找必须生成一个声明才能编译程序。查找范围中的名称会查找该名称的所有声明，但有一个例外，称为“结构黑客”或“类型/非类型隐藏”：在同一范围内，某些名称的出现可能会引用声明不是typedef的类/结构体/联合体/枚举类型，而所有其他名称相同的名称都指向相同的变量，非静态数据成员（自C ++ 14以来）或枚举器，或者它们全部请参阅可能重载的函数或函数模板名称。在这种情况下，没有错误，但类型名称不能查找（代码必须使用详细的类型说明符来访问它）。

对于函数和函数模板名称，名称查找可以将多个声明和相同的名称进行关联，且可通过参数依赖查找获得额外的声明。应用模板参数推导，将这些声明传递给重载决策，由它选择使用哪个声明。成员访问规则，只有在名称查找和重载决策之后才会被考虑。

### 1.7.1 限定名称查找

A qualified name is a name that appears on the right hand side of the scope resolution operator ：： (see also qualified identifiers). A qualified name may refer to a

* class member (including static and non-static functions, types, templates, etc)
* namespace member (including another namespace)
* enumerator

If there is nothing on the left hand side of the ：：, the lookup considers only declarations made in the global namespace scope (or introduced into the global namespace by a using declaration). This makes it possible to refer to such names even if they were hidden by a local declaration：

#include <iostream>

int main() {

struct std{};

std::cout << "fail\n"; // Error: unqualified lookup for 'std' finds the struct

::std::cout << "ok\n"; // OK: ::std finds the namespace std

}

Before name lookup can be performed for the name on the right hand side of ：：, lookup must be completed for the name on its left hand side (unless a decltype expression is used, or there is nothing on the left). This lookup, which may be qualified or unqualified, depending on whether there's another ：： to the left of that name, considers only namespaces, class types, enumerations, and templates whose specializations are types：

struct A {

static int n;

};

int main() {

int A;

A::n = 42; // OK: unqualified lookup of A to the left of :: ignores the variable

A b; // Error: unqualified lookup of A finds the variable A

}

### 1.7.2 非限定名称查找

For an unqualified name, that is a name that does not appear to the right of a scope resolution operator ::, name lookup examines the scopes as described below, until it finds at least one declaration of any kind, at which time the lookup stops and no further scopes are examined. (Note: lookup from some contexts skips some declarations, for example, lookup of the name used to the left of :: ignores function, variable, and enumerator declarations, lookup of a name used as a base class specifier ignores all non-type declarations)

对于非限定名称，这是一个名称，它不会出现在范围解析操作符::的右侧，名称查找将按照以下所述检查范围，直到找到至少一个任何类型的声明，此时查找停止 （注意：从某些上下文中查找会跳过一些声明，例如，查找用于:: ignores函数，变量和枚举器声明左侧的名称，查找用作基类的名称 说明符忽略所有非类型声明）

For the purpose of unqualified name lookup, all declarations from a namespace nominated by a using directive appear as if declared in the nearest enclosing namespace which contains, directly or indirectly, both the using-directive and the nominated namespace.

为了进行非限定名称查找，来自由using指令外观提名的命名空间的所有声明，如同在最近的封闭名称空间中声明的那样，它直接或间接地包含using-directive和指定的名称空间。

Unqualified name lookup of the name used to the left of the function-call operator (and, equivalently, operator in an expression) is described in argument-dependent lookup.

在函数左侧使用名称的非限定名称查找 - 调用操作符（和表达式中的等价操作符）在参数相关查找中进行了描述。

#### 1.7.2.1 全局作用域

对于在全局（顶层命名空间）作用域中使用的名称，在任何函数，类或用户声明的名称空间之外，将检查使用该名称之前的全局作用域：

|  |
| --- |
| int n = 1; // 声明n  int x = n + 1; // OK:查找发现::n  int z = y - 1; // Error: 查找失败  int y = 2; // 声明y |

#### 1.7.2.2 命名空间

对于在任何函数或类之外的用户声明命名空间中使用的名称，在使用名称之前搜索此命名空间，然后在声明此命名空间之前包括此命名空间的命名空间等，直到到达全局命名空间。

|  |
| --- |
| int n = 1; // 声明  namespace N {  int m = 2;  namespace Y {  int x = n; // OK, 查找到::n  int y = m; // OK, 查找到::N::m  int z = k; // Error:查找失败  }  int k = 3;  } |

#### 1.7.2.3 命名空间之外的定义

对于在命名空间外定义命名空间成员变量时使用的名称，查找过程与命名空间内使用的名称相同：

|  |
| --- |
| namespace X {  extern int x; // 声明，没有定义  int n = 1; // 发现第一个n  };  int n = 2; // 发现第二个n  int X::x = n; // 发现X::n，设置X::x为1 |

#### 1.7.2.4 非成员函数定义

对于在函数定义中使用的名称，无论是在其函数体中，还是作为默认参数的一部分，其中函数是用户声明的或全局的命名空间的成员时，在使用名称之前将搜索使用该名称的块，然后搜索包含该块的封闭块，直到到达作为函数体的块。然后搜索声明该函数的命名空间，直到使用名称的函数的定义（不一定是声明），然后搜索外围命名空间等。

|  |
| --- |
| namespace A {  namespace N {  void f();  int i=3; // 发现第3个（如果第2个不存在）  }  int i=4; // 发现第4个（如果第3个不存在）  }  int i=5; // 发现第5个（如果第4个不存在）    void A::N::f() {  int i = 2; // 发现第2个（如果第一个不存在）  while(true) {  int i = 1; // 发现第1个，查找结束  std::cout << i;  }  }    // int i; // 不被发现    namespace A {  namespace N {  // int i; // 不被发现  }  } |

#### 1.7.2.5 类定义

对于在类定义中任何地方出现的名称，除了成员函数体内部，成员函数的默认参数，成员函数的异常说明，默认成员初始值或嵌套类定义内部（包括嵌套类派生），搜索以下范围：

* 使用名称的类体内直到使用的地方
* 基类的整体，当没有发现声明时，递归到它们的基类
* 如果这个类是嵌套的，外围类的主体直到这个类的声明和外围类的基类的整个主体。
* 如果这个类是局部的，或者嵌套在局部类中，那么这个类的定义的块作用域直到定义点
* 如果这个类是一个命名空间的成员，或者嵌套在一个命名空间成员的类中，或者是一个命名空间成员的函数中的一个局部类，则搜索命名空间的范围直到类的定义，外围类或函数。如果查找由友邻声明引入的名称：在这种情况下，只考虑最内层的外围命名空间，否则会继续查找外围命名空间直到全局范围内。

|  |
| --- |
| namespace M {  // const int i = 1; // 永远不被发现  class B {  // const const int i = 3; // found 3nd (but later rejected by access check)  };  }  // const int i = 5; // found 5th  namespace N {  // const int i = 4; // found 4th  class Y : public M::B {  // static const int i = 2; // found 2nd  class X {  // static const int i = 1; // found 1st  int a[i]; // use of i  // static const int i = 1; // never found  };  // static const int i = 2; // never found  };  // const int i = 4; // never found  }  // const int i = 5; // never found |

#### 1.7.2.6 注入类名

对于在该类或模板或派生类的定义中使用的类或类模板的名称，非限定名称查找会查找被定义的类，就好像该名称是由成员声明（带有公共成员访问权限）引入的那样。 有关更多详细信息，请参阅[注入类名](#_9.2_injected-class-name)。

#### 1.7.2.7 成员函数定义

For a name used inside a member function body, a default argument of a member function, exception specification of a member function, default member initializer, or inside a nested class definition (including names of the bases from which the nested class is derived), the scopes searched are the same as in class definition, except that the entire scope of the class is considered, not just the part prior to the declaration that uses the name. For nested classes the entire body of the enclosing class is searched.

|  |
| --- |
| class B {  // int i; // found 3rd  };  namespace M {  // int i; // found 5th  namespace N {  // int i; // found 4th  class X : public B {  // int i; // found 2nd  void f();  // int i; // found 2nd as well  };  // int i; // found 4th  }  }  // int i; // found 6th  void M::N::X::f()  {  // int i; // found 1st  i = 16;  // int i; // never found  }  namespace M {  namespace N {  // int i; // never found  }  } |

Either way, when examining the bases from which the class is derived, the following rules, sometime referred to as [dominance in virtual inheritance](https://en.wikipedia.org/wiki/Dominance_(C%2B%2B)), are followed:

A member name found in a sub-object B hides the same member name in any sub-object A if A is a base class sub-object of B. (Note that this does not hide the name in any additional, non-virtual, copies of A on the inheritance lattice that aren't bases of B: this rule only has an effect on virtual inheritance.) Names introduced by using-declarations are treated as names in the class containing the declaration. After examining each base, the resulting set must either include declarations of a static member from subobjects of the same type, or declarations of non-static members from the same subobject（C++11之前）。

A lookup set is constructed, which consists of the declarations and the subobjects in which these declarations were found. Using-declarations are replaced by the members they represent and type declarations, including injected-class-names are replaced by the types they represent. If C is the class in whose scope the name was used, C is examined first. If the list of declarations in C is empty, lookup set is built for each of its direct bases Bi (recursively applying these rules if Bi has its own bases). Once built, the lookup sets for the direct bases are merged into the lookup set in C as follows（C++11之后）

* if the set of declarations in Bi is empty, it is discarded
* if the lookup set of C built so far is empty, it is replaced by the lookup set of Bi
* if every subobject in the lookup set of Bi is a base of at least one of the subobjects already added to the lookup set of C, the lookup set of Bi is discarded.
* if every subobject already added to the lookup set of C is a base of at least one subobject in the lookup set of Bi, then the lookup set of C is discarded and replaced with the lookup set of Bi
* otherwise, if the declaration sets in Bi and in C are different, the result is an ambiguous merge: the new lookup set of C has an invalid declaration and a union of the subobjects ealier merged into C and introduced from Bi. This invalid lookup set may not be an error if it is discarded later.
* otherwise, the new lookup set of C has the shared declaration sets and the union of the subobjects ealier merged into C and introduced from Bi

|  |
| --- |
| struct X { void f(); };  struct B1: virtual X { void f(); };  struct B2: virtual X {};  struct D : B1, B2 {  void foo() {  X::f(); // OK, calls X::f (qualified lookup)  f(); // OK, calls B1::f (unqualified lookup)  // C++98 rules: B1::f hides X::f, so even though X::f can be reached from D  // through B2, it is not found by name lookup from D.  // C++11 rules: lookup set for f in D finds nothing, proceeds to bases  // lookup set for f in B1 finds B1::f, and is completed  // merge replaces the empty set, now lookup set for f in C has B1::f in B1  // lookup set for f in B2 finds nothing, proceeds to bases  // lookup for f in X finds X::f  // merge replaces the empty set, now lookup set for f in B2 has X::f in X  // merge into C finds that every subobject (X) in the lookup set in B2 is a base  // of every subobject (B1) already merged, so the B2 set is discareded  // C is left with just B1::f found in B1  // (if struct D : B2, B1 was used, then the last merge would \*replace\* C's  // so far merged X::f in X because every subobject already added to C (that is X)  // would be a base of at least one subobject in the new set (B1), the end  // result would be the same: lookup set in C holds just B1::f found in B1)  }  }; |

Unqualified name lookup that finds static members of B, nested types of B, and enumerators declared in B is unambiguous even if there are multiple non-virtual base subobjects of type B in the inheritance tree of the class being examined:

|  |
| --- |
| struct V { int v; };  struct A {  int a;  static int s;  enum { e };  };  struct B : A, virtual V { };  struct C : A, virtual V { };  struct D : B, C { };    void f(D& pd) {  ++pd.v; // OK: only one v because only one virtual base subobject  ++pd.s; // OK: only one static A::s, even though found in B and in C  int i = pd.e; // OK: only one enumerator A::e, even though found in B and C  ++pd.a; // error, ambiguous: A::a in B and A::a in C  } |

#### 1.7.2.8 友邻函数定义

For a name used in a friend function definition inside the body of the class that is granting friendship, unqualified name lookup proceeds the same way as for a member function. For a name used in a friend function which is defined outside the body of a class, unqualified name lookup proceeds the same way as for a function in a namespace.

|  |
| --- |
| int i = 3; // found 3rd for f1, found 2nd for f2  struct X {  static const int i = 2; // found 2nd for f1, never found for f2  friend void f1(int x)  {  // int i; // found 1st  i = x; // finds and modifies X::i  }  friend int f2();  // static const int i = 2; // found 2nd for f1 anywhere in class scope  };  void f2(int x) {  // int i; // found 1st  i = x; // finds and modifies ::i  } |

1.7.2.9 友邻函数声明

For a name used in the declarator of a friend function declaration that friends a member function from another class, if the name isn't a part of any template argument, the unqualified lookup first examines the entire scope of the member function's class. If not found in that scope (or if the name is a part of a template argument), the lookup continues as if for a member function of the class that is granting friendship

|  |
| --- |
| // the class whose member functions are friended  struct A {  typedef int AT;  void f1(AT);  void f2(float);  template <class T> void f3();  };    // the class that is granting friendship  struct B {  typedef char AT;  typedef float BT;  friend void A::f1(AT); // lookup for AT finds A::AT  friend void A::f2(BT); // lookup for BT finds B::BT  friend void A::f3<AT>(); // lookup for AT finds B::AT  }; |

#### 1.7.2.9 默认参数

For a name used in a default argument in a function declaration, or name used in the expression part of a member-initializer of a constructor, the function parameter names are found first, before the enclosing block, class, or namespace scopes are examined:

|  |
| --- |
| class X {  int a, b, i, j;  public:  const int& r;  X(int i): r(a), // initializes X::r to refer to X::a  b(i), // initializes X::b to the value of the parameter i  i(i), // initializes X::i to the value of the parameter i  j(this->i) // initializes X::j to the value of X::i  { }  }    int a;  int f(int a, int b = a); // error: lookup for a finds the parameter a, not ::a  // and parameters are not allowed as default arguments |

#### 1.7.2.10 静态数据成员定义

For a name used in the definition of a static data member, lookup proceeds the same way as for a name used in the definition of a member function.

|  |
| --- |
| struct X {  static int x;  static const int n = 1; // found 1st  };  int n = 2; // found 2nd.  int X::x = n; // finds X::n, sets X::x to 1, not 2 |

#### 1.7.2.11 枚举器声明

For a name used in the initializer part of the enumerator declaration, previously declared enumerators in the same enumeration are found first, before the unqualified name lookup proceeds to examine the enclosing block, class, or namespace scope.

|  |
| --- |
| const int RED = 7;  enum class color {  RED,  GREEN = RED+2, // RED finds color::RED, not ::RED, so GREEN = 2  BLUE = ::RED+4 // qualified lookup finds ::RED, BLUE = 11  }; |

#### 1.7.2.12 try-catch语句

对于函数try-catch中使用的名称，查找过程就像在函数体的最外面块的开头使用的名称一样（特别是函数参数是可见的，但是在最外面的块不是）

|  |
| --- |
| int n = 3; // 发现第3个  int f(int n = 2) // 发现第2个  try {  int n = -1; // 永远不能找到  } catch(...) {  // int n = 1; // 发现第一个  assert(n == 2); // 查找n找到函数参数f  throw;  } |

#### 1.7.2.13 重载运算符

For an operator used in expression (e.g., operator+ used in a+b), the lookup rules are slightly different from the operator used in an explicit function-call expression such as operator+(a,b): when parsing an expression, two separate lookups are performed: for the non-member operator overloads and for the member operator overloads (for the operators where both forms are permitted). Those sets are then merged with the built-in operator overloads on equal grounds as described in overload resolution. If explicit function call syntax is used, regular unqualified name lookup is performed:

|  |
| --- |
| struct A {};  void operator+(A, A); // user-defined non-member operator+    struct B {  void operator+(B); // user-defined member operator+  void f ();  };    A a;    void B::f() // definition of a member function of B  {  operator+(a,a); // error: regular name lookup from a member function  // finds the declaration of operator+ in the scope of B  // and stops there, never reaching the global scope  a + a; // OK: member lookup finds B::operator+, non-member lookup  // finds ::operator+(A,A), overload resolution selects ::operator+(A,A)  } |

1.7.2.14 模板定义

For a non-dependent name used in a template definition, unqualified name lookup takes place when the template definition is examined. The binding to the declarations made at that point is not affected by declarations visible at the point of instantiation. For a dependent name used in a template definition, the lookup is postponed until the template arguments are known, at which time ADL examines function declarations with external linkage (until C++11) that are visible from the template definition context as well as in the template instantiation context, while non-ADL lookup only examines function declarations with external linkage (until C++11) that are visible from the template definition context (in other words, adding a new function declaration after template definition does not make it visible except via ADL). The behavior is undefined if there is a better match with external linkage in the namespaces examined by the ADL lookup, declared in some other translation unit, or if the lookup would have been ambiguous if those translation units were examined. In any case, if a base class depends on a template parameter, its scope is not examined by unqualified name lookup (neither at the point of definition nor at the point of instantiation).

|  |
| --- |
| void f(char); // first declaration of f    template<class T>  void g(T t) {  f(1); // non-dependent name: lookup finds ::f(char) and binds it now  f(T(1)); // dependent name: lookup postponed  f(t); // dependent name: lookup postponed  // dd++; // non-dependent name: lookup finds no declaration  }    enum E { e };  void f(E); // second declaration of f  void f(int); // third declaration of f  double dd;    void h() {  g(e); // instantiates g<E>, at which point  // the second and the third uses of the name 'f'  // are looked up and find ::f(char) (by lookup) and ::f(E) (by ADL)  // then overload resolution chooses ::f(E).  // This calls f(char), then f(E) twice  g(32); // instantiates g<int>, at which point  // the second and the third uses of the name 'f'  // are looked up and find ::f(char) only  // then overload resolution chooses ::f(char)  // This calls f(char) three times  }    typedef double A;  template<class T> class B {  typedef int A;  };  template<class T> struct X : B<T> {  A a; // lookup for A finds ::A (double), not B<T>::A  }; |

Note: see dependent name lookup rules for the reasoning and implications of this rule.

## 1.8 存储模型和数据竞争

Defines the semantics of computer memory storage for the purpose of the C++ abstract machine.

C++程序可用的内存是一个或多个连续的字节序列。内存中的每个字节都有一个唯一的地址。

### 1.8.1 字节

字节byte是内存中最小的可寻址单元。是由连续的位组成，足够保存任何UTF-8代码单元（256个不同的值）和基本执行字符集的任何成员（96个字符要求是单字节）。与C相似，C++支持大小为8位或者更大的字节。

The types char, unsigned char, and signed char use one byte for both storage and value representation. The number of bits in a byte is accessible as CHAR\_BIT or std：：numeric\_limits<unsigned char>：：digits.

类型char，unsigned char，signed char使用一个字节进行存储和值表达。一个字节中的

### 1.8.2 存储单元

存储单元是：

* 标量类型的对象 (运算符类型，指针类型，枚举类型或std：：nullptr\_t)
* 或者最大的非零的连续位域

注意：C++语言多个特征，诸如引用和虚函数（virtual），可能会引入程序不可访问的存储单元，而是由实现管理的。

struct S {

char a; // 存储位置 #1

int b : 5; // 存储位置 #2

int c : 11, // 存储位置 #2 (连续的)

: 0,

d : 8; // 存储位置 #3

struct {

int ee : 8; // 存储位置 #4

} e;

} obj; // 对象 'obj' 由独立的4个存储位置

### 1.8.3 线程和数据竞争

通过std：：thread：：thread，std：：async，或者其他方法调用顶层函数，实现线程的执行，其执行过程是一个流控过程。

在程序中，任何线程潜在地可以访问任何对象（带有automatic和thread-local存储周期的对象可以被其它线程使用指针或引用进行访问）。

不同的线程同时访问不同的存储单元，对其进行读写，没有干扰，也没有同步的要求。

当一个表达式的值写入一块存储单元的同时，另一表达式值也在读或者修改同一存储单元，这样就会发生冲突。有两个冲突的值的程序就会发生“数据竞争”除非

* 两个冲突的求值表达式在相同的线程或者同一个信号处理函数里执行，或
* 两个冲突的求值表达式都是原子操作（见：std：：atomic），或
* 两个冲突的求值表达式执行是顺序的（见std：：memory\_order）

如果发生数据竞争，程序的行为是未定义状态。

（可以使用std：：mutex和另一个线程进行同步，来避免数据竞争问题）

int cnt = 0;

auto f = [&]{cnt++;};

std::thread t1{f}, t2{f}, t3{f}; // 未定义的行为

std::atomic<int> cnt{0};

auto f = [&]{cnt++;};

std::thread t1{f}, t2{f}, t3{f}; // OK

### 1.8.4 内存顺序

当线程从内存位置读取值时，它可能会看到同一线程写入的初始值，也有可能是另一个线程写入的值。请参阅std :: memory\_order了解对其他线程可见的线程写入顺序的详细信息。

### 1.8.5 Forward progress

**1 无阻塞（Obstruction freedom）**

只有当调用标准库函数不会发生阻塞的线程，执行一个无锁的原子操作时，这个执行才能保证完成。（所有标准库无锁操作都是非阻塞的。）

**2 无锁**

当一个或多个无锁的原子操作函数同时运行时，它们中至少一个被保证执行（所有标准库的无锁操作都是无锁的—这是由实现保证的，确保它们不能被别的线程无限锁住，例如被同时占据cache line）。

**3. 执行保证**

在合法的C++程序里，每一个线程终将会做下面的某一步：

* 终止
* 调用IO库函数
* 访问易失性[泛左值](#_4.1.2.1_泛左值（glvalue）)
* 执行原子操作或同步操作

没有任何线程可以在没有执行这些可观察的行为中任一个，而永远执行的。

请注意，这意味着具有无限递归或无限循环的程序（无论是作为for语句还是通过循环goto或其它方式实现）都具有未定义的行为。这允许编译器删除所有没有可观察行为的循环，而不必证明它们最终会终止。

A thread is said to make progress if it performs one of the execution steps above (I/O, volatile, atomic, or synchronization), blocks in a standard library function, or calls an atomic lock-free function that does not complete because of a non-blocked concurrent thread.

如果一个线程执行上面的一个执行步骤（I/O，易失性，原子或同步），标准库函数中的块，或者调用一个无法完成的原子无锁函数（因为 一个非阻塞的并发线程。

Concurrent forward progress（C++17之后）

If a thread offers concurrent forward progress guarantee, it will make progress (as defined above) in finite amount of time, for as long as it has not terminated, regardless of whether other threads (if any) are making progress.

如果一个线程提供并发远期进度保证，只要它没有终止，它将在有限的时间内（如上所定义的）进展，而不管其他线程（如果有的话）是否正在取得进展。

The standard encourages, but doesn't require that the main thread and the threads started by std：：thread offer concurrent forward progress guarantee.

Parallel forward progress（C++17之后）

If a thread offers parallel forward progress guarantee, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step (I/O, volatile, atomic, or synchronization), but once this thread has executed a step, it provides concurrent forward progress guarantees (this rule describes a thread in a thread pool that executes tasks in arbitrary order)

Weakly parallel forward progress（C++17之后）

If a thread offers weakly parallel forward progress guarantee, it does not guarantee to eventually make progress, regardless of whether other threads make progress or not.

Such threads can still be guaranteed to make progress by blocking with forward progress guarantee delegation： if a thread P blocks in this manner on the completion of a set of threads S, then at least one thread in S will offer a forward progress guarantee that is same or stronger than P. Once that thread completes, another thread in S will be similarly strengthened. Once the set is empty, P will unblock.

The parallel algorithms from the C++ standard library block with forward progress delegation on the completion of an unspecified set of library-managed threads.

无锁进度保证的概念理解：[stackoverflow.com上的一篇文章](https://stackoverflow.com/questions/45907210/lock-free-progress-guarantees)

有趣的是，我发现很多程序员错误地认为“无锁”意味着“没有互斥的并发编程”。通常，还有一个相关的误解，即编写无锁代码的目的是为了获得更好的并发性能。当然，无锁的正确定义实际上是关于进度保证(progress guarantees)。无锁算法保证至少有一个线程能够向前进展（forward progress），而不管其他线程在做什么。

这意味着无锁算法永远不会有一个线程依赖于另一个线程才能继续的代码。例如，无锁代码不能存在线程A设置flag，然后线程B循环等待线程A取消flag的情况。这样的代码基本上实现了一个锁（或者我会称之为伪装的互斥锁）。

然而，其他情况更加微妙，并且在某些情况下，我真的不能确定算法是否符合无锁规则，因为“making progress”的概念有时对我来说似乎是主观的。

一个这样的例子是（好评，afaik）并发库liblfds。我正在研究在liblfds中实现一个多生产者/多客户有界的队列 - 这个实现非常简单，但我无法确定它是否应该被视为无锁。

相关算法在lfds711\_queue\_bmm\_enqueue.c中。Liblfds使用自定义原子和内存屏障（memory barriers），但该算法足够简单，可让我用一段描述。

队列本身是一个有界的连续数组（环形缓存）。有一个共享的read\_index和write\_index。队列中的每个位置都包含一个用于用户数据的字段和一个序列号，它基本上就像一个纪元计数器。 （这可以避免ABA问题）。

PUSH算法如下：

* 以原子方式LOAD write\_index
* Attempt to reserve a slot in the queue at write\_index % queue\_size using a CompareAndSwap loop that attempts to set write\_index to write\_index + 1.
* If the CompareAndSwap is successful, copy the user data into the reserved slot.
* Finally, update the sequence\_index on the slot by making it equal to write\_index + 1.

The actual source code uses custom atomics and memory barriers, so for further clarity about this algorithm I've briefly translated it into (untested) standard C++ atomics for better readability, as follows:

|  |
| --- |
| bool mcmp\_queue::enqueue(void\* data)  {  int write\_index = m\_write\_index.load(std::memory\_order\_relaxed);  for (;;)  {  slot& s = m\_slots[write\_index % m\_num\_slots];  int sequence\_number = s.sequence\_number.load(std::memory\_order\_acquire);  int difference = sequence\_number - write\_index;  if (difference == 0)  {  if (m\_write\_index.compare\_exchange\_weak(  write\_index,  write\_index + 1,  std::memory\_order\_acq\_rel  ))  {  break;  }  }  if (difference < 0) return false; // queue is full  }  // Copy user-data and update sequence number  //  s.user\_data = data;  s.sequence\_number.store(write\_index + 1, std::memory\_order\_release);  return true;  } |

Now, a thread that wants to POP an element from the slot at read\_index will not be able to do so until it observes that the slot's sequence\_number is equal to read\_index + 1.

Okay, so there are no mutexes here, and the algorithm likely performs well (it's only a single CAS for PUSH and POP), but is this lock-free? The reason it's unclear to me is because the definition of "making progress" seems murky when there is the possibility that a PUSH or POP can always just fail if the queue is observed to be full or empty.

But what's questionable to me is that the PUSH algorithm essentially reserves a slot, meaning that the slot can never be POP'd until the push thread gets around to updating the sequence number. This means that a POP thread that wants to pop a value depends on the PUSH thread having completed the operation. Otherwise, the POP thread will always return false because it thinks the queue is EMPTY. It seems debatable to me whether this actually falls within the definition of "making progress".

Generally, truly lock-free algorithms involve a phase where a pre-empted thread actually tries to ASSIST the other thread in completing an operation. So, in order to be truly lock-free, I would think that a POP thread that observes an in-progress PUSH would actually need to try and complete the PUSH, and then only after that, perform the original POP operation. If the POP thread simply returns that the queue is EMPTY when a PUSH is in progress, the POP thread is basically blocked

until the PUSH thread completes the operation. If the PUSH thread dies, or goes to sleep for 1,000 years, or otherwise gets scheduled into oblivion, the POP thread can do nothing except continuously report that the queue is EMPTY.

So does this fit the defintion of lock-free? From one perspective, you can argue that the POP thread can always make progress, because it can always report that the queue is EMPTY (which is at least some form of progress I guess.) But to me, this isn't really making progress, since the only reason the queue is observed as empty is because we are blocked by a concurrent PUSH operation.

So, my question is: is this algorithm truly lock-free? Or is the index reservation system basically a mutex in disguise?

## 1.9 转译阶段

编译器处理C++源文件的过程可以看做下面几个阶段：

### 1.9.1 阶段1

1）源文件的每一个字节都会被映射为基本字符集中的字符。尤其是，依赖于操作系统的行结束符都会被换行符取代。基本字符集包含96个字符：

* 5 种空白字符 (空格， 水平制表符，垂直制表符，换页符，换行符)；
* 10 个数字[0-9]；
* 26 个英文字母的大小写；
* 29 标点符号：\_ { } [ ] # ( ) < > % ： ； . ? \* + - / ^ & | ~ ! = , \ " '

2）源文件中，任何不能由基本字符集映射的字符，使用通用字符取代（即使用\转义符号进行转义）或者由其编译器作相应的处理。

3）三字符序列被相应的单字符表达方式替代。（until C++17）

### 1.9.2 阶段2

1）一旦在某一行的结尾发现反斜杠“\”（后面紧跟换行符），符号“\”和换行符被删掉，把源文件的两行连接成一行。这是一次单程操作；如果一行结束时，后面紧跟两个反斜杠“\”字符和空行，那么是不会把这三行组成一个新行的。如果在这个阶段，出现通用字符（\uXXX），这种行为未被定义。

如果这个阶段之后，非空的源文件没有以换行符结束（不论是本来就没有换行符，或它以反斜杠符号结束），C++11之前没有定义这种行为，C++11之后会添加一个换行符。

### 1.9.3 阶段3

1） 源文件被解析成注释，空白字符序列（空格，水平制表符，换行符，垂直制表符，和换页符），预处理符号，如下所示：

* 头文件名称，例如 <iostream> 或者 "myfile.h"
* 标识符
* 预处理数字
* 字符或字符串，包括用户自定义的 (C++11之后添加)
* 操作符和标点符号(包括可替换的符号), 如+，<<=，new，<%，##，and（&&）
* 不属于其它种类的单个非空白字符

2） 在阶段1和阶段2里，任何实施过转换的由双引号“”包含的原始字符串都会被恢复。（C++11之后）

3） 注释被一个空格替代

换行符被保留，没有明确说明，非换行符的空白字符序列是否会被整合成一个空格字符。

### 1.9.4 阶段4

1）预处理程序被执行；

2）递归遍历阶段1到阶段4，用#include指令引入每一个文件；

3）在这个阶段结束时，所有源文件的预处理指令被移除。

### 1.9.5 阶段5

1）由源文件转换而来的字符文字和字符串文字里的所有字符都被设为可执行字符集（有可能是如UTF-8一样的多字节字符集）

2） 字符文字和非原始字符串文字里的转义序列和通用字符被展开并转换为可执行字符集。如果由通用字符指定的字符不是可执行字符集里的成员，结果是编译器指定，但是保证不是一个null字符（广义上）

注意：这个阶段的转换执行，在某些编译器的实现里，可以由命令行选项进行控制：gcc和clang使用选项-finput-charset 指定源字符集的编码格式；-fexec-charset和-fwide-exec-charset，指定字符串和字符文字的可执行字符集的编码格式（没有编码前缀）。（C++11之后）

### 1.9.6 阶段6

相邻字符串文字链接为新的字符串文字。

### 1.9.7 阶段7

编译阶段：每个预处理符号被转换为一个符号。这些符号被从语法上和语义上分析，然后转换为一个翻译单元。

### 1.9.8 阶段8

每一个翻译单元被检查，找出要求模板实例化的列表，包括哪些被要求显式实例化的模板。找到那些被要求实例化的模板，然后执行，产生实例化单元。

### 1.9.9 阶段9

翻译单元，实例化单元，库组件满足外部引用的被集成都一个程序镜像，它包括在执行环境里执行时需要的信息。

### 1.9.10 注意

一些编译器不实现实例化单元（也被称为模板仓库或模板注册表）且在阶段7编译每一个模板实例，然后存储代码到对应的目标文件中，然后链接器在阶段9把这些编译后的目标文件集合为一个可执行文件。

### 1.9.11 参考

* C++11 standard (ISO/IEC 14882：2011)：
* 2.2 Phases of translation [lex.phases]
* C++98 standard (ISO/IEC 14882：1998)：
* 2.1 Phases of translation [lex.phases]

## 1.10 main() 函数

Ddd

## 1.11 布局

### 1.11.1 标准布局类型

指定一个类型是标准布局类型。标准布局类型对于与使用其他编程语言编写的代码进行配合很有用。

请注意，该标准没有定义具有此名称的命名要求或概念。这是由核心语言定义的类型类别。只是作为概念出现在这里。

#### 1.11.1.1 要求

1. 所有非静态数据成员都具有相同的访问控制；

2. 没有虚函数或虚基类；

3. 没有引用类型的非静态数据成员；

4. 所有非静态数据成员和基类本身都是标准布局类型；

5. 另外，

|  |  |
| --- | --- |
| 1. 没有具有非静态数据成员的基类，或在最底层派生类里没有非静态数据成员且最多只有一个基类具有非静态数据成员；（C++14之前） 2. 没有和第一个非静态数据成员相同类型的基类（参见[空基优化](#_12.6_空基优化)）； | C++14之前 |
| （1）不能具有两个相同类型的基类子对象（很可能不是直接具有）  struct Q {}；  struct S ： Q { }；  struct T ： Q { }；  struct U ： S, T { }； // 非标准布局类： 有2个Q类型的基类  （2）在同一个类中声明所有非静态数据成员和位域（另外，全部在派生或其它的基类中）  struct B { int i； }； // 标准布局类  struct C ： B { }； // 标准布局类  struct D ： C { }； // 标准布局类  struct E ： D { char ： 4； }； // 非标准布局类  6.  （1）对于非union类型，没有任何一个基类子对象和第一个非静态数据成员具有相同的类型（请参阅[空基优化](#_12.6_空基优化)），递归地，如果该非静态数据成员是非union类类型，也没有任何一个基类子对象和该数据成员的第一个非静态数据成员具有相同的类型；或如果该非静态数据成员是union类型，也没有任何一个基类子对象和该数据成员的所有非静态数据成员具有相同的类型；再或者，如果该非静态数据成员是数组类型，也没有任何一个基类子对象和该数据成员的数组元素具有相同的类型。  （2）对于联合类型，没有任何一个基类子对象和任何一个非静态数据成员具有相同的类型；递归地，也不能和非union类类型的每一个成员的第一个非静态数据成员具有相同类型；也不能和union类型的所有成员的所有非静态数据成员具有相同类型；也不能和数组类型的所有非静态数据成员的元素类型具有相同类型。  （3）对于数组类型，作为数组元素的类型，如果数组元素具有非联合类类型，则递归地返回数组元素的第一个非静态数据成员，如果它是数组元素的任何非静态数据成员 具有联合类型，或者如果数组元素具有数组类型，则为数组元素的元素类型等。 | C++14之后 |

#### 1.11.1.2 属性

查看[标准布局](#_9.3.1.2_标准布局)。

## 1.12 文字类型（LiteralType）

Specifies that a type is a literal type. Literal types are the types of constexpr variables and they can be constructed, manipulated, and returned from constexpr functions.

Note, that the standard doesn't define a named requirement or concept with this name. This is a type category defined by the core language. It is included here as concept only for consistency.

指定一个类型是一个文字类型。文字类型是constexpr变量的类型，它们可以被构造，操作并从constexpr函数返回。

注意：标准中没有定义具有此名称的命名要求或概念。这是由核心语言定义的类型类别。在这里介绍，只是为了一致性的概念。

### 1.12.1 要求

文字类型可以是下面的任一个：

* 可能的cv限定的void（如果那样，constexpr函数可以返回void）（C++14）；
* 标量类型；
* 引用类型；
* 文字类型的数组；
* 可能的cv限定的class类型，其具有下面的属性：
  + 具有没有意义的析构函数，
  + is either
    - an aggregate type,
    - a type with at least one constexpr (possibly template) constructor that is not a copy or move constructor,
    - a closure type(since C++17)
  + for unions, at least one non-static data member is of non-volatile literal type, (since C++17)
  + for non-unions, all non-static data members and base classes are of non-volatile literal types.(since C++17)
  + all non-static data members and base classes are of non-volatile literal types.
  + (until C++17)

### 1.12.2 举例

|  |
| --- |
| #include <iostream>  #include <stdexcept>  class conststr  {  const char\* p;  std::size\_t sz;  public:  template<std::size\_t N>  constexpr conststr(const char(&a)[N]) : p(a), sz(N - 1) {}    constexpr char operator[](std::size\_t n) const  {  return n < sz ? p[n] : throw std::out\_of\_range("");  }  constexpr std::size\_t size() const { return sz; }  };    constexpr std::size\_t countlower(conststr s, std::size\_t n = 0,  std::size\_t c = 0)  {  return n == s.size() ? c :  s[n] >= 'a' && s[n] <= 'z' ? countlower(s, n + 1, c + 1) :  countlower(s, n + 1, c);  }    // output function that requires a compile-time constant, for testing  template<int n>  struct constN  {  constN() { std::cout << n << '\n'; }  };    int main()  {  std::cout << "the number of lowercase letters in \"Hello, world!\" is ";  constN<countlower("Hello, world!")>(); // implicitly converted to conststr  } |

输出结果：

|  |
| --- |
| the number of lowercase letters in "Hello, world!" is 9 |

# 2 C++关键字

## 2.1 各类关键字

### 2.1.1 using关键字

命名空间使用using指令，using声明用于命名空间成员；

类成员可以使用using声明进行声明；

类型别名和别名模板声明（C++11之后）；

# 3 预处理器

## 3.1 条件预编译

#if - #ifdef - #ifndef

## 3.2 文本宏替换

#define - # - ## - #include

## 3.3其它预处理指令

#error - #pragma - #line

# 4 表达式

An expression is a sequence of operators and their operands, that specifies a computation.

Expression evaluation may produce a result (e.g., evaluation of 2+2 produces the result 4) and may generate side-effects (e.g. evaluation of std::printf("%d",4) prints the character '4' on the standard output).

表达式是一系列运算符及其操作数，用于指定计算。

表达式评估可能会产生一个结果（例如，评估2 + 2会产生结果4），并可能产生副作用（例如，评估std :: printf（“％d”，4））在标准上打印字符'4'输出）。

## 4.0 概述

value categories (lvalue, rvalue, glvalue, prvalue, xvalue) classify expressions by their values

值类别（左值，右值，glvalue，prvalue，xvalue）通过它们的值对表达式进行分类

order of evaluation of arguments and subexpressions specify the order in which intermediate results are obtained

参数和子表达式的评估顺序指定了获得中间结果的顺序

operator precedence defines the order in which operators are bound to their arguments

运算符优先级定义了运算符绑定到它们的参数的顺序

alternative representations are alternative spellings for some operators

备选表述是一些运营商的备选拼写

operator overloading makes it possible to specify the behavior of the operators with user-defined classes.

运算符重载使得可以使用用户定义的类来指定运算符的行为。

### 4.0.1 操作符

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 常用操作符 | | | | | | |
| 赋值 | 自加、自减 | 算术 | 逻辑 | 比较 | 成员访问 | 其它 |
| a = b  a += b  a -= b  a \*= b  a /= b  a %= b  a &= b  a |= b  a ^= b  a <<= b  a >>= b | ++a  --a  a++  a-- | +a  -a  a + b  a - b  a \* b  a / b  a % b  ~a  a & b  a | b  a ^ b  a << b  a >> b | !a  a && b  a || b | a == b  a != b  a < b  a > b  a <= b  a >= b  a <=> b | a[b]  \*a  &a  a->b  a.b  a->\*b  a.\*b | a(...)  a, b  ? : |
| 特殊操作符 | | | | | | |
| static\_cast-将一种类型转换为另一种相关类型  dynamic\_cast-在继承层次结构中转换  const\_cast-添加或删除cv限定符  reinterpret\_cast-将类型转换为不相关的类型  C-style cast-C类型转换，编译器会自动解释为static\_cast、const\_cast、reinterpret\_cast中的一个  new-创建动态存储对象  delete-析构先前new创建的对象并释放获取的内存区域  sizeof -查询类型的大小  sizeof...-查询参数包的大小（C++11之后）  typeid-查询类型信息  noexcept-检查表达式是否可以抛出异常（C++11之后）  alignof-查询类型的对齐要求（C++11之后） | | | | | | |

### 4.0.2 转换

标准转换-可以实现类型隐式转换

const\_cast

static\_cast

dynamic\_cast

reinterpret\_cast

显式转换-使用C风格转换符号和功能符号的显式转换转换

用户自定义转换-可以指定从用户定义的类转换

### 4.0.3 内存分配

new-动态分配内存

delete-动态释放内存

### 4.0.4 其它

constant expressions 编译时计算且用于上下文中（模板参数、数组大小等等）

sizeof

alignof

typeid

throw-expression

### 4.0.5 基本表达式

任何操作符的操作数可以是其它表达式或基本表达式（例如，在1 + 2 \* 3中，操作符+的操作数是子表达式2 \* 3和基本表达式1）。

基本表达式有以下几种：

1. 文字（例，2或“Hello,world”）
2. 声明的未限定标识符（例，n或cout）
3. 声明的限定标识符（例，std::string::npos）
4. Lambda表达式(C++11)
5. Fold-expressions (C++17)
6. Requires-expressions (C++20)

括号中的任何表达式也被分类为基本表达式：这保证括号的优先级高于任何运算符。括号保存数值，类型和值类别。

### 4.0.6 文字

文字是代表嵌入到源代码中的常量值的C++程序的符号。

integer literals are decimal, octal, hexadecimal or binary numbers of integer type.

character literals are individual characters of type char, char16\_t, char32\_t, or wchar\_t

floating-point literals are values of type float, double, or long double

string literals are sequences of characters of type const char[], const char16\_t[], const char32\_t[], or const wchar\_t[]

boolean literals are values of type bool, that is true and false

nullptr is the pointer literal which specifies a null pointer value (since C++11)

user-defined literals are constant values of user-specified type (since C++11)

### 4.0.7 不计算表达式

The operands of the operators typeid, sizeof, noexcept, and decltype (since C++11) are expressions that are not evaluated (unless they are polymorphic glvalues and are the operands of typeid), since these operators only query the compile-time properties of their operands. Thus, std::size\_t n = sizeof(std::cout << 42); does not perform console output.

操作符typeid，sizeof，noexcept和decltype（自C ++ 11以来）的操作数是不被评估的表达式（除非它们是多态的glvalues并且是typeid的操作数），因为这些操作符仅查询编译时属性 的操作数。 因此，std :: size\_t n = sizeof（std :: cout << 42）; 不执行控制台输出。

The unevaluated operands are considered to be full expressions even though they are syntactically operands in a larger expression (for example, this means that sizeof(T()) requires an accessible T::~T)

未被评估的操作数被认为是完整的表达式，即使它们是在较大表达式中的语法操作数（例如，这意味着sizeof（T（））需要可访问的T ::〜T）（C++14）。

The requires-expressions are also unevaluated expressions.

(since C++20)

require-expressions也是未评估的表达式。（C++20后）

### 4.0.8 弃值表达式

弃值表达式是仅使用其副作用的表达式，求值的结果被抛弃。这类表达式包括任何表达式语句的完整表达式，内嵌逗号运算符的左侧参数，或者转换为void类型的cast表达式所使用的参数。

Array-to-pointer and function-to-pointer conversions are never applied to the value calculated by a discarded-value expression. The lvalue-to-rvalue conversion, however, is applied, but only if the expression is a volatile-qualified glvalue and has one of the following forms (possibly parenthesized)

数组到指针和函数到指针的转换绝不会应用于由丢弃值表达式计算的值。 然而，左值到右值的转换被应用，但是只有当表达式是一个符合volatile限定的泛左值并且具有下列形式之一（可能被括起来）

* id表达式
* 数组下标表达式
* 类成员访问表达式
* indirection
* 成员指针
* 条件表达式，其第二或第三操作数是这类表达式之一，
* 逗号表达式，其中右操作数是这些表达式之一。

|  |  |
| --- | --- |
| In addition, if the expression is of class type, a volatile copy-constructor is required to initialize the resulting rvalue temporary. | C++17之前 |
| If the expression is a prvalue (after any lvalue-to-rvalue conversion that might have taken place), temporary materialization occurs. If the original glvalue is of volatile-qualified class type, a volatile copy-constructor is required to initialize the resulting rvalue temporary.  Compilers may issue warnings when an expression other than cast to void discards a value declared [[nodiscard]] | C++17之后 |

## 4.1 值类别

每个C++表达式（具有操作数，文字常量，变量等的操作符）都具有两个独立的属性：一个类型和一个值类别。每个表达式都有一些非引用类型，每个表达式都恰好属于三个基本值类别之一：纯右值（prvalue），消亡值（xvalue），左值（lvalue），定义如下：

1. 泛左值是一个表达式，它的求值结果决定了是对象、位域还是函数；
2. 纯右值是下面两种情况中的任一种，
   1. 对操作符的操作数进行求值（这样的纯右值没有结果对象），或，
   2. 初始化对象或者[位域](#_9.9.1_位域)（这样的纯右值具有结果对象）。所有的类和数组类型的纯右值都有结果对象，即使它们被放弃。在某些情况下，temporary materialization 会发生，产生临时对象作为结果对象。
3. 消亡值是一种泛左值，表示对象或位域的资源能够被重利用；
4. 左值就是除了消亡值之外的泛左值；
5. 右值是纯右值或消亡值。

总结：

（1）glvalue（泛左值） = lvalue（传统意义上的左值）+ xvalue（消亡值，通过右值引用产生）

（2）rvalue（传统意义上的右值） = prvalue（纯右值） + xvalue

C++11之前只有左值和右值的概念：lvalue，rvalue。左值可以取地址，右值不能取地址。

但是C++11之后又划分的更加详细了，分为左值（lvalue），纯右值（prvalue）还有将亡值（xvalue），关系如下：

注意：该分类标准经历了过去C++标准修订版的重大变化，请参阅下面的历史部分以了解详细信息。

### 4.1.1 基本类型

#### 4.1.1.1 左值

下面的表达式是左值表达式：

* 变量，函数或数据成员的名称，不管类型，如std::cin或std::endl。即使变量的类型是右值引用，由其名称组成的表达式也是一个左值表达式；
* 函数调用或重载运算符表达式，其返回类型是左值引用，如std::getline（std::cin，str），std::cout << 1，str1 = str2或++it；
* a = b, a += b, a %= b，及所有其它内嵌赋值和复合赋值表达式；
* ++a和--a，内置前自加和前自减表达式；
* \*p，内嵌间接表达式（如指针）；
* a[n]和p[n]是内嵌下标表达式，除非a是数组右值（C++11以后）；
* a.m，对象成员表达式，除非m是成员枚举器或非静态成员函数，或者a是右值且m是非引用类型的非静态成员函数。
* p->m，内嵌指针表达式的成员，除非m是成员枚举器或非静态成员函数；
* a.\*mp，指向对象成员的指针表达式，其中a是左值，mp是指向数据成员的指针；
* p->\*mp，指向指针成员的内置指针表达式，其中mp是指向数据成员的指针；
* a，b，内置的逗号表达式，此时逗号表达式最右的值是左值，这儿，b是一个左值；
* a ? b : c，三元条件表达式（例如，当两者都是相同类型的左值时，请参见[定义](#_4.4.5_call,_comma,)细节）；
* 字符串，例如“Hello，world！”；特别注意，C++规定字符串是左值；
* 转换为左值引用类型的cast表达式，例如static\_cast <int&>(x)；
* 函数调用或重载运算符表达式，其返回类型是函数的右值引用;（C++11之后）；
* 转换为函数类型的右值引用的转换表达式，如static\_cast<void(&&)(int)>(x)。（C++11之后）。

属性：

* 和泛左值一样（如下）。
* 可以使用左值的地址：&++i和&std::endl是有效的表达式。
* 可修改的左值可以用作内置赋值和复合赋值操作符的左操作数。
* 左值可以用来初始化左值引用;这会将新名称与表达式标识的对象关联起来。

#### 4.1.1.2 纯右值

下面的表达式是纯右值表达式：

* 文字（除了字符串文字常量），例如42，true或nullptr；
* 函数调用或重载操作符表达式，其返回类型是非引用，比如，str.substr(1,2)，str1+str2,或it++；
* a++和a--，内置后自增和后自减表达式；
* a+b，a%b，a&b，a<<b，和其它所有算术表达式；
* a&&b，a||b，!a，内置逻辑表达式；
* a<b，a==b，a>=b，和其它内置比较表达式；
* &a，内置取地址符；
* a.m，对象成员表达式，在这，m是成员枚举或非静态成员函数；或a是右值，m是非引用类型的非静态数据成员；（C++11之前）
* p->m，内置指针成员表达式，m是成员枚举，或非静态成员函数；
* a.\*mp，指向对象的成员的指针表达式，mp是指向成员函数的指针，或者a是右值且mp是指向数据成员的指针。（C++11之前）
* p->\*mp，内置指向指针成员的指针表达式，mp是指向成员函数的指针；
* a,b，内置逗号“，”表达式，在这，b是右值；
* a?b:c，条件表达式（细节参见定义）；
* 非引用类型的转换表达式，比如static\_cast<double>(x)，std::string{}，或(int)42；
* this指针；
* 枚举；
* [lambda 表达式](#_7.4_Lambda表达式(C++11))，例如[](int x){ return x \* x; }；（C++11之后）
* [requires-expression](#_10.12_Constraints_and)，比如requires (T i) { typename T::type; }；
* [概念的特化](#_10.12_Constraints_and)，比如EqualityComparable<int>；（C++20之后）

属性：

* 与右值相同（见下面）。
* 纯右值表达式不能是多态：它标识的对象的动态类型始终是表达式的类型；
* 既不是类，也不是数组的纯右值不能被进行cv限定。（注意：函数调用或转换表达式可能导致不是class类型且被cv限定的类型的纯右值，但是cv限定符立即被剔除）
* 纯右值不能拥有[不完整的类型](#_1.4.6_不完整类型)（除了类型void，看下面，或者当在[声明类型说明符](#_5.11_decltype-auto(C++11))里使用时）。

注意：纯右值是传统右值的一部分，纯右值是表达式产生的中间值，不能取地址。

#### 4.1.1.3 消亡值（xvalue）

下面的值是消亡值表达式：

* 函数调用和重载操作符表达式，其返回值是对象的右值引用，比如std::move(x)；
* a[n]，内置的下标表达式，在这里，一个操作数是数组右值；
* a.m，对象成员表达式，a是右值且m是非引用类型的非静态数据成员；
* a.\*mp，指向对象成员的指针表达式，a是右值且且mp是指向数据成员的指针；
* a ? b : c，条件表达式，详见定义；
* 对象类型的右值引用的类型转换，比如static\_cast<char&&>(x)；
* 在temporary materialization之后，指定临时对象的任何表达式；（C++17）

属性：

* 和右值相同（参见下面的内容）；
* 和泛左值相同（参见下面的内容）；

特殊地，像所有右值一样，消亡值可以绑定到右值引用，且向所有泛左值一样，消亡值可以是多态的，且非class型的消亡值可以被cv限定。

### 4.1.2 混合类型

#### 4.1.2.1 泛左值（glvalue）

泛左值表达式是左值或消亡值。

属性：

一个泛左值可以通过左值到右值，数组到指针或函数到指针的[隐式转换](#_4.8.1_implicit_-)而转换为纯右值。

泛左值可能是[多态](#_1.5.1.3_多态对象)的：它识别的对象的[动态类型](#_1.4.5_动态类型)不一定是表达式的静态类型。

在表达式允许的情况下，泛左值可能具有不完整的类型。

#### 4.1.2.2 右值（rvalue）

右值表达式既可以是纯右值也可以是消亡值。

属性：

（1）右值地址不可取：&int()，&i++ 和 &std::move(x)是非法的；

（2）右值不能作为赋值运算符的“左手边”算子；

（3）右值可以被用来[初始化const左值引用](#_6.7_引用初始化)，在这种情况下，右值对象的生命周期延长到引用结束。

（4）右值也可以被用来[初始化右值引用](#_6.7_引用初始化)，在这种情况下，右值对象的生命周期延长到引用结束。（C++11之后）

（5）当被用作函数参数，且该函数有[2个重载版本](#_7.7_Overload_resolution)可用，一个采用右值引用作为参数，另一个采用常量的左值引用作为参数，右值被绑定到右值引用重载。（因而，如果copy和move构造函数都存在，右值参数就会调用move构造函数，拷贝和移动赋值操作符也是这样）。（C++11之后）

### 4.1.3 特殊类型

1. Pending member function call

表达式a.mf和p->mf，在这里，mf是非静态成员函数；和表达式a.\*mfp和p->\*mfp，在这里，mfp是成员函数的指针，都被分类为纯右值表达式，但是它们不能用来初始化引用，作为函数参数，或者除了作为函数调用的左边参数以外的任何其它目的。（例如，(p->\*mfp)(args)）。

2. void

Function call expressions returning void, cast expressions to void, and throw-expressions are classified as prvalue expressions, but they cannot be used to initialize references or as function arguments. They can be used in discarded-value contexts (e.g. on a line of its own, as the left-hand operand of the comma operator, etc.) and in the return statement in a function returning void. In addition, throw-expressions may be used as the second and the third operands of the conditional operator ?:.

函数调用表达式返回void，将表达式转换为void以及throw-expressions被分类为prvalue表达式，但它们不能用于初始化引用或作为函数参数。 它们可以在丢弃值的上下文中使用（例如，在它自己的一行上，作为逗号操作符的左侧操作数等），并且在函数的返回语句中返回void。 另外，throw-expressions可以用作条件运算符的第二个和第三个操作数？：。

Void expressions have no result object. (since C++17)

3. 位域

An expression that designates a bit field (e.g. a.m, where a is an lvalue of type struct A { int m: 3; }) is an lvalue expression: it may be used as the left-hand operand of the assignment operator, but its address cannot be taken and a non-const lvalue reference cannot be bound to it. A const lvalue reference can be initialized from a bit-field lvalue, but a temporary copy of the bit-field will be made: it won't bind to the bit field directly.

指定位域（egam，其中a是类型struct A {int m：3;}的左值）的表达式是左值表达式：它可以用作赋值运算符的左侧操作数，但其地址 不能被采用，并且非const的左值引用不能被绑定到它。 常量左值引用可以从位域左值初始化，但会创建位域的临时副本：它不会直接绑定到位域。

### 4.1.4 变更历史

CPL

The programming language CPL was first to introduce value categories for expressions: all CPL expressions can be evaluated in "right-hand mode", but only certain kinds of expression are meaningful in "left-hand mode". When evaluated in right-hand mode, an expression is regarded as being a rule for the computation of a value (the right-hand value, or rvalue). When evaluated in left-hand mode an expression effectively gives an address (the left-hand value, or lvalue). "Left" and "Right" here stood for "left of assignment" and "right of assignment".

C

The C programming language followed a similar taxonomy, except that the role of assignment was no longer significant: C expressions are categorized between "lvalue expressions" and others (functions and non-object values), where "lvalue" means an expression that identifies an object, a "locator value"[4].

C++98

Pre-2011 C++ followed the C model, but restored the name "rvalue" to non-lvalue expressions, made functions into lvalues, and added the rule that references can bind to lvalues, but only references to const can bind to rvalues. Several non-lvalue C expressions became lvalue expressions in C++.

C++11

With the introduction of move semantics in C++11, value categories were redefined to characterize two independent properties of expressions[5]:

* has identity: it's possible to determine whether the expression refers to the same entity as another expression, such as by comparing addresses of the objects or the functions they identify (obtained directly or indirectly);
* can be moved from: move constructor, move assignment operator, or another function overload that implements move semantics can bind to the expression.

In C++11, expressions that:

* have identity and cannot be moved from are called lvalue expressions;
* have identity and can be moved from are called xvalue expressions;
* do not have identity and can be moved from are called prvalue ("pure rvalue") expressions;
* do not have identity and cannot be moved from are not used[6].

The expressions that have identity are called "glvalue expressions" (glvalue stands for "generalized lvalue"). Both lvalues and xvalues are glvalue expressions.

The expressions that can be moved from are called "rvalue expressions". Both prvalues and xvalues are rvalue expressions.

C++17

In C++17, copy elision was made mandatory in some situations, and that required separation of prvalue expressions from the temporary objects initialized by them, resulting in the system we have today. Note that, in contrast with the C++11 scheme, prvalues are no longer moved from.

## 4.2 操作数计算顺序

C++是一个注重效率的语言，标准不指定一些表达式的求值顺序就是为了让编译器能做尽可能多的优化，即便要牺牲掉例如 i=i++ 这样表达式的正确性。

在C++中，几乎所有操作符的操作数的求值顺序（包括函数调用表达式中函数参数的计算顺序以及任何表达式中子表达式的计算顺序）都未指定。编译器可以按任何顺序计算操作数，也许再计算一次该表达式又是另一种顺序。

对于这条规则，有下面不适用的地方，

除了下面指出的地方，在C++中没有从左到右或从右到左的计算顺序概念。这儿不应和操作符的从左到右或从右到左结合顺序的概念混淆：由于操作符+的从左到右的结合性，表达式f1()+f2()+f3()被解析为(f1()+f2())+f3()，但是，在运行时，对于函数f3的调用可能是先调用，也可能最后调用，还有可能是在f1和f2调用之间。

### 4.2.1 提前序列化规则（C++11之后）

#### 4.2.1.1定义

**1. 求值**

编译器对每个表达式或子表达式执行两种计算（两者都是可选的）：

* 值计算：计算表达式返回的值。这涉及到对象身份的确定（泛左值求值，例如，如果表达式返回某个对象的引用）或者读取先前赋值给对象的值（纯右值求值，例如，如果表达式返回一个数字，或某个其它值）。
* 副作用：访问由volatile型的泛左值指定的对象（读/写），修改对象，调用I/O库函数，或调用这些操作的函数。

**2. 顺序**

“sequenced-before”是相同的线程内不同计算之间一种不对称，可传递，成对的关系。

* 如果A在B之前，那么A的计算在B开始之前完成。
* 如果A的计算顺序在B之后，那么B的计算就在A开始之前完成。
* A也不在B的前面，B也不在A的前面，那么有两种可能：
  + A和B的求值顺序是不确定的：它们可以以任何顺序执行并且可以重叠（在单个执行线程内，编译器可以交叉组成A和B的CPU指令）。
  + 对A和B的求值有不确定的顺序：它们可以按任意顺序执行，但可能不重叠：A在B之前完成，或B在A之前完成。下次再计算相同的表达式，顺序可能就是相反的。

#### 4.2.1.2 规则

1） 每个完整表达式的值计算和副作用（side effect），即

* 未计算的操作数
* 常量表达式
* 整个初始化器，包括任何逗号分隔的构成表达式
* 非临时对象的生命周期结束时产生的析构函数调用
* 一个表达式不是另一个完整表达式的一部分（例如整个表达式语句，for / while循环控制的表达式，if / switch的条件表达式，return语句等），

包括应用于表达式结果的隐式转换，对临时对象的析构函数调用，默认成员初始化（初始化聚合时）以及涉及函数调用的每个其他语言结构，在每个值计算和side effect之前被序列化。

2） 操作数对任何操作符的值计算（但不包括副作用）在操作符结果的值计算之前序列化（但不包括其副作用）。

3） 在调用一个函数时（无论函数是否是内联的，是否使用显式函数调用语法），与任何参数表达式或指定被调用函数的后缀表达式相关的每个值计算和副作用，在执行被调用函数的主体中的每个表达式或语句之前进行序列化。

4）内嵌后自增和后自减运算符的值计算在其副作用之前序列化。

5）内嵌前自增和前自减运算符的副作用在其值计算之前序列化。（由于定义为复合赋值而产生的隐式规则）

6）内嵌逻辑AND运算符&&和内嵌逻辑OR运算符||的第一个（左）参数的每个值计算和副作用在第二个（右）参数的每个值计算和副作用之前序列化。

7）条件运算符？:中的第1个表达式相关的每个值计算和副作用在第2个或第3个表达式关联的值计算和副作用之前被序列化。

8）内嵌赋值运算符和所有内嵌复合赋值运算符的副作用（左参数的修改）在左和右两个参数的值计算（但不是副作用）之后进行序列化，并且是在赋值表达式的值计算之前序列化（即，在返回引用修改的对象之前）

9）内嵌逗号运算符的第一个（左）参数的每个值计算和副作用在第二个（右）参数的每个值计算和副作用之前序列化。

10）在列表初始化中（使用大括号括起来，用逗号分割的初始化列表中），给定初始化子句的每个值计算和副作用在跟随在它之后的任何初始化子句关联的值计算和副作用之前进行序列化。

11）在另一个未被确定地序列化的函数调用之前或之后，这个函数都不会序列化（程序必须表现得好像构成不同函数调用的CPU指令没有交错，即使函数被内联）。

**规则11有一个例外：由标准库算法在std :: execution :: par\_unseq执行策略下执行的函数调用不被序列化，并且可以被任意交错。（C++17之后）**

12) The call to the allocation function (operator new) is indeterminately sequenced with respect to (until C++17)sequenced before (since C++17) the evaluation of the constructor arguments in a new-expression

13) When returning from a function, copy-initialization of the temporary that is the result of evaluating the function call is sequenced-before the destruction of all temporaries at the end of the operand of the return statement, which, in turn, is sequenced-before the destruction of local variables of the block enclosing the return statement.（C++14之后）

|  |  |
| --- | --- |
| 14) In a function-call expression, the expression that names the function is sequenced before every argument expression and every default argument.  15) In a function call, value computations and side effects of the initialization of every parameter are indeterminately sequenced with respect to value computations and side effects of any other parameter.  16) Every overloaded operator obeys the sequencing rules of the built-in operator it overloads when called using operator notation.  17) In a subscript expression E1[E2], every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2  18) In a pointer-to-member expression E1.\*E2 or E1->\*E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2 (unless the dynamic type of E1 does not contain the member to which E2 refers)  19) In a shift operator expression E1<<E2 and E1>>E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2  20) In every simple assignment expression E1=E2 and every compound assignment expression E1@=E2, every value computation and side-effect of E2 is sequenced before every value computation and side effect of E1  21) Every expression in a comma-separated list of expressions in a parenthesized initializer is evaluated as if for a function call (indeterminately-sequenced) | C++17之后 |

#### 4.2.1.3 未定义行为

1) 如果标量对象上的副作用相对于同一标量对象上的另一副作用而不被序列化，则行为是不确定的。

|  |
| --- |
| i = ++i + 2; // 未定义行为（直到C++11）  i = i++ + 2; // 未定义行为（直到C++17）  f(i = -2, i = -2); // 未定义行为（直到C++17）  f(++i, ++i); // 未定义行为（直到C++17），C++17之后未指明  i = ++i + i++; // 未定义行为 |

2) 如果对标量对象的副作用和使用相同标量对象的值进行的值计算没有序列化的关系，则行为不确定。

|  |
| --- |
| cout << i << i++; // 未定义行为（直到C++17）  a[i] = i++; // 未定义行为（直到C++17）  n = ++i + i; // 未定义行为 |

### 4.2.2 序列点规则（C++11之前）

#### 4.2.2.1 定义

对表达式的计算可能会产生副作用，即：访问由易变的左值指定的对象，修改对象，调用库I/O函数或调用执行任何这些操作的函数。

序列点是执行序列中的一个点，序列中先前计算的所有副作用都已完成，并且没有后续计算的副作用开始。

#### 4.2.2.2 规则

1) 在每一个完整表达式的末尾有一个序列点（通常以分号表示）。

2）在调用一个函数时（函数是否内联以及是否使用函数调用语法），在执行函数体内任何表达式或语句之前，所有函数参数（如果有）的计算之后，有一个序列点。

3）拷贝函数的返回值之后和执行函数外的任何表达式之前，有一个序列点。

4）一旦函数开始执行，就不会计算任何正在发生调用的函数的表达式，直到被调用函数的执行完成（函数不能交错）为止。

5）在计算以下四种表达式时，使用内置（非重载）运算符，评估表达式a后有一个序列点。

|  |
| --- |
| a && b  a || b  a ? b : c  a,b |

#### 4.2.2.3 未定义行为

1）在前一个和下一个序列点之间，一个标量对象经过表达式的计算后，最多修改其存储值一次，否则是[未定义行为](#_12.3_未定义行为)。

|  |
| --- |
| i = ++i + i++; // 未定义行为  i = i++ + 1; // 未定义行为(直到C++17)  i = ++i + 1; // 未定义行为(直到C++11)  ++ ++i; // 未定义行为(直到C++11)  f(++i, ++i); // 未定义行为(直到C++17)  f(i = -1, i = -1); // 未定义行为(直到C++17) |

2）在前一个和下一个序列点之间，必须仅访问通过评估表达式修改的标量对象的先前值，以确定要存储的值。 如果以任何其他方式访问它，则行为未定义。

## 4.3 常量表达式

定义了一个编译时计算的表达式。

这样的表达式可以用作非类型模板参数，数组大小以及需要常量表达式的其它上下文，例如，

|  |
| --- |
| int n = 1;  std::array<int, n> a1; // 错误：n不是常量表达式  const int cn = 2;  std::array<int, cn> a2; // 正确：cn是常量表达式 |

### 4.3.1 核心常量表达式

核心常数表达式是不会对以下任何一项进行计算的表达式：

1. [this指针](#_9.9.2_this指针)，例外就是，在常量表达式函数或者常量表达式构造函数里被计算为表达式的一部分。

2. 函数调用表达式中，调用的函数（或构造函数）没有被声明为[常量表达式](#_5.10.2_解释)

|  |
| --- |
| constexpr int n = std::numeric\_limits<int>::max(); // OK: max()是常量表达式  constexpr int m = std::time(nullptr); // 错误: std::time()不是常量表达式 |

3. 调用声明为常量表达式，但是没有定义的函数，

4. 调用不能满足常量表达式函数或构造函数的条件的函数

5. 超出了执行时定义的限制的表达式

6. 表达式的计算会导致任何形式的核心语言未定义行为（包括有符号整数溢出，零除，指针算术外数组边界等）。 未检测到标准库未定义行为是未指定的。

|  |
| --- |
| constexpr double d1 = 2.0/1.0; // OK  constexpr double d2 = 2.0/0.0; // Error: 没有定义  constexpr int n = std::numeric\_limits<int>::max() + 1; // Error: 溢出  int x, y, z[30];  constexpr auto e1 = &y - &x; // Error: 未定义  constexpr auto e2 = &z[20] - &z[3]; // OK  constexpr std::bitset<2> a;  constexpr bool b = a[2]; // UB, but unspecified if detected |

7. lambda表达式（C++17之后）

8. 左值到右值的隐式转换，除非左值，

1. 具有整型或枚举类型，并引用一个完整的非易失性常量对象，该对象用常量表达式进行初始化

|  |
| --- |
| int main() {  const std::size\_t tabsize = 50;  int tab[tabsize]; // OK: tabsize是常量表达式    std::size\_t n = 50;  const std::size\_t sz = n;  int tab2[sz]; // error: sz 不是常量表达式  // 因为sz未使用常量表达式进行初始化  } |

1. 引用字符串文字的非易失性泛左值
2. 具有文字类型，并且引用常量表达式或其具有non-mutable的子对象的非易失性对象
3. 具有文字类型，且引用生命周期在该表达式计算中开始的非易失性对象。

9. an lvalue-to-rvalue implicit conversion or modification applied to a non-active member of a union or its subobject (even if it shares a common initial sequence with the active member)

10. an invocation of an implicitly-defined copy/move constructor or copy/move assignment operator for a union whose active member (if any) is mutable, unless the lifetime of the union object began within the evaluation of this expression

11. (since C++17) an assignment expression or invocation of an overloaded assignment operator that would change the active member of a union

12. an id-expression referring to a variable or a data member of reference type, unless it was initialized with a constant expression or its lifetime began within the evaluation of this expression

13. conversion from cv void\* to any pointer-to-object type

14. dynamic\_cast

15. reinterpret\_cast

16. pseudo-destructor call

17. (until C++14) an increment or a decrement operator

18. (since C++14) modification of an object, unless the object has non-volatile literal type and its lifetime began within the evaluation of the expression

|  |
| --- |
| constexpr int incr(int& n) {  return ++n;  }  constexpr int g(int k) {  constexpr int x = incr(k); // 错误: incr(k)不是核心常量表达式，因为k生命周期在incr(k)之外  return x;  }  constexpr int h(int k) {  int x = incr(k); // OK: x 没有被要求使用核心常量表达式进行初始化  return x;  }  constexpr int y = h(1); // OK: 使用数值2初始化y，h(1)是核心常量表达式，  // 因为k的生命周期在表达式h(1)内 |

19. a typeid expression applied to a glvalue of polymorphic type

20. a new-expression or a delete-expression

21. (since C++20) a three-way comparison comparing pointers that do not point to the same complete object or to any subobject thereof

22. an equality or relational operator when the result is unspecified

23. (until C++14) an assignment or a compound assignment operator

24. throw表达式

25. 在lambda表达式中，引用this或在lambda之外定义的变量，如果该引用是使用了[ODR规则](#_1.6.3_ODR-use)。

|  |
| --- |
| void g() {  const int n=0;  constexpr int j=\*&n; // OK: lambda表达式之外  [=]{ constexpr int i=n; // OK: 没有使用唯一定义规则，这里不会捕获  constexpr int j=\*&n;// 不合法： '&n' would be an odr-use of 'n'.  };  } |

## 4.4 操作符

### 4.4.1 赋值

### 4.4.2 算术

#### 4.4.2.1 解释

#### 4.4.2.2 转换

#### 4.4.2.3 溢出

#### 4.4.2.4 浮点环境

#### 4.4.2.5 Floating-point contraction

#### 4.4.2.6 一元运算符

#### 4.4.2.7 加减

#### 4.4.2.8 乘除

#### 4.4.2.9 位逻辑

#### 4.4.2.10 移位

### 4.4.3 自增和自减

### 4.4.4 逻辑

### 4.4.5 比较

### 4.4.6 成员访问

### 4.4.7 其它操作符（函数调用，逗号，条件表达式）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 操作符名称 | 语法 | 重载 | 示例（Class T） | |
| 类定义里面 | 类定义之外 |
| 函数调用 | a(a1, a2) | Yes | R T::operator()(Arg1 &a1, Arg2 &a2,...); | N/A |
| 逗号 | a,b | Yes | T2& T::operator,(T2 &b); | T2& operator,(const T &a, T2 &b); |
| 条件操作符 | a?b:c | No | N/A | N/A |

函数调用运算符为任何对象提供函数语义。

三目条件运算符检查第一个表达式的布尔值，并根据结果决定返回第二个或第三个表达式。

#### 4.4.7.1 内嵌函数调用（已完成）

语法：

|  |
| --- |
| E(A1,A2,A3,…) |

在这里，

1. E是命名函数的表达式；
2. A1,A2,A3,…是一个可能为空的任意表达式列表，这里的表达式不能再使用逗号，避免歧义。

命名函数的表达式E，可以是，

1. 引用函数的左值表达式；
2. 函数指针；
3. 显式地选择成员函数的类成员访问表达式；
4. 隐式类成员访问表达式，例如在另一个成员函数里使用的成员函数名。

由表达式E指定的函数名称（或成员函数名称）能够被重载，根据重载决策规则决定使用哪个重载版本。

如果E指定了成员函数，而成员函数是虚的（virtual），在这种情况下，在运行时就会使用动态分配调用final重载关键字。

对于求值顺序，表达式E和其它所有的表达式A1，A2，A3等可以以任意顺序，非序列化的方式进行求值，彼此之间没有要求。（C++17之前）

但是，在C++17之后，表达式E的顺序要在参数表达式之前完成求值，参数表达式A1，A2，A3等之间以任意顺序求值，彼此之间以不确定的顺序排序。

|  |
| --- |
| #include <cstdio>  struct S  {  int f1(double d) {  return printf("%f \n", d); // 变量参数函数调用  }  int f2() {  return f1(7); // 成员函数调用，等同于this->f1()  // int型参数转化成double型  }  };  void f() {  puts("function called"); // 函数调用  }  int main()  {  f(); // 函数调用  S s;  s.f2(); // 成员函数调用  } |

在GCC 5.2(C++17)上编译结果：

|  |
| --- |
| function called  7.000000 |

#### 4.4.7.2 内嵌逗号操作符（已完成）

语法如下：

E1,E2

在上面的逗号表达式中，表达式E1被求值，但是结果被抛弃（但如果它具有class类型，直到包含的完整表达式结束才会被销毁），并且在表达式E2的求值之前完成E1的[副作用](#_4.2.1_提前序列化规则（C++11之后）)操作。

逗号表达式的结果的类型、值、和值类别恰好是第二个操作数E2的类型、值、和值类别。如果E2是一个临时表达式，逗号表达式的结果就是一个临时对象。如果E2是一个位域，结果就是一个位域。

由多个逗号隔开的列表，比如函数参数列表（f(a,b,c)）和初始化列表int a[] = {1,2,3}，它们中的逗号不是操作符。如果需要在这样的上下文中使用，必须用括号括起来，f(a,(n++,n+b),c)。

|  |
| --- |
| #include <iostream>  int main()  {  int n = 1;  int m = (++n, std::cout << "n = " << n << '\n', ++n, 2\*n);  std::cout << "m = " << (++m, m) << '\n';  } |

在GCC 5.2(C++11)上编译结果：

|  |
| --- |
| n = 2  m = 7 |

#### 4.4.7.3 条件操作符

语法

E1 ? E2 : E3

条件操作符的第一个操作数E1被求值，然后执行上下文转换为bool型。在第一个操作数完成求值和所有的副作用之后，如果结果是true，则第二个操作数E2被求值。如果结果是false，则第三个操作数被求值。

条件表达式的类型和值类别根据下面的规则决定：

如果E2或E3的类型是void，则下面必须有一个是true，否则就是语法错误：

如果E2或者E3，有一个但不都是抛出异常表达式。条件操作符的结果就是另一个表达式的类型和值类别。如果另一个表达式是位域，则结果就是位域。

### 4.4.8 sizeof操作符

查询对象和类型的大小。在需要获取对象的实际大小使用。

#### 4.4.8.1 语法

|  |  |
| --- | --- |
| sizeof(*type*) | （1） |
| Sizeof *expression* | （2） |

两个版本都返回一个 std::size\_t 类型的常量。

#### 4.4.8.2 解释

1. 返回*type*的对象表示的字节数。
2. 返回当*expression*求值时所返回的类型的对象表示的字节数。

#### 4.4.8.3 注意

取决于计算机架构，byte可能具有8位或更多位，精确的位数记录于 CHAR\_BIT 。

sizeof(char)，sizeof(signed char)，和sizeof(unsigned char)总是返回1。

sizeof不能用于函数类型，不完整类型，或者位表示的泛左值。

当应用于引用类型时，其结果是被引用类型的大小。

当应用于类类型时，其结果是该类的对象的大小与这种对象放入数组时所需的额外填充的大小的总和。

当应用于空类时，总是返回1。

当应用于某个表达式时，sizeof 并不对表达式进行求值，并且即便表达式代表多态对象，其结果也是该表达式的静态类型的大小。不进行左值向右值、数组向指针和函数向指针转换。（不过，它在形式上对纯右值实参进行临时量实质化： sizeof 确定其结果对象的大小。(C++17 起)）

#### 4.4.8.4 示例

该示例假设64位的指针和32位的int型。

|  |
| --- |
| #include <iostream>    struct Empty {};  struct Base { int a; };  struct Derived : Base { int b; };  struct Bit { unsigned bit: 1; };    int main()  {  Empty e;  Derived d;  Base& b = d;  [[maybe\_unused]] Bit bit;  int a[10];  std::cout << "size of empty class: " << sizeof e << '\n' // out->1  << "size of pointer: " << sizeof &e << '\n' // out->4  // << "size of function: " << sizeof(void()) << '\n' // error  // << "size of incomplete type: " << sizeof(int[]) << '\n' // error  // << "size of bit field: " << sizeof bit.bit << '\n' // error  << "size of array of 10 int: " << sizeof(int[10]) << '\n' // out->40  << "size of array of 10 int (2): " << sizeof a << '\n' // out->40  << "length of array of 10 int: " << ((sizeof a)/(sizeof \*a)) <<'\n' // out->10  << "length of array of 10 int (2): " << ((sizeof a)/(sizeof a[0]))<<'\n' // out->10  << "size of the Derived: " << sizeof d << '\n' // out->8  << "size of the Derived through Base: " << sizeof b << '\n'; // out->4    } |

### 4.4.9 alignof(C++11)

### 4.4.10 new

创建并初始化拥有动态[存储周期](#_5.7.3_存储期)的对象，即生命周期不受创建它们时所在的作用域限制的对象。

#### 4.4.10.1 语法

|  |  |
| --- | --- |
| ::(可选) new (placement\_params)(可选) (type) initializer(可选) | （1） |
| ::(可选) new (placement\_params)(可选) type initializer(可选) | （2） |

1. 尝试创建[typeid](#_4.4.12_typeid)-type所表示的类型的一个对象，它可以是数组类型，可以包含类型指定符auto（C++11之后）或者占位符类型decltype(auto)（C++17之后）；
2. 同上，但type不能包含括号：

|  |
| --- |
| new int(\*[10])(); // 错误：分析成 (new int) (\*[10]) ()  new (int (\*[10])()); // 正确：分配一个具有10个函数指针的数组 |

另外，没有括号的type是贪婪的，它将包含任何可能是声明符一部分的记号：

|  |
| --- |
| new int + 1; // 正确：分析成 (new int) + 1 ，增加 new int 所返回的指针  new int \* 1; // 错误：分析成 (new int\*) (1) |

注意：如果在type中使用auto的话，initializer不是可选的，要根据它推导出替代auto的类型：

|  |
| --- |
| auto p = new auto('c'); // 创建类型为char的对象，p 为char\*的指针 |

#### 4.4.10.2 解释

new 表达式尝试申请存储空间，并在已申请的存储空间上，尝试构造并初始化为一个未命名的对象，或未命名的对象数组。new表达式返回一个指向构造的对象或者对象数组的纯右值指针。（也就是说明，它不能被赋值）

如果type是数组类型，除了第一维之外，所有的维数都应指定为正整形的常量表达式（C++14之前）、为std::size\_t的常量表达式（C++14之后）但是第一维可以是任何能够转换为std::size\_t的表达式。这是唯一的直接创建大小在运行时定义的数组的方法，这种数组被称为动态数组。

|  |
| --- |
| int n = 42;  double a[n][5]; // 错误  auto p1 = new double[n][5]; // 正确  auto p2 = new double[5][n]; // 错误 |

下列情况中指定第一维的表达式是错误的：

* 表达式是非类类型且其值在转换到 std::size\_t 前是负的；
* the expression is of class type and its value after user-defined conversion function and before the second standard conversion is negative;
* 表达式的值大于某些实现定义极限；
* 该值小于[大括号括起来的初始化器](#_6.5_Aggregate_initialization)提供的数组元素的数量(包括字符串变量的终止符'\0')。

如果出于上述任何原因，第一维中的值是错误的，

* 如果在转换成std::size\_t 之后，第一维是一个核心常量表达式，程序是不规范的。(发出编译时错误)；
* 否则，如果调用的分配函数是非异常抛出的，则new表达式返回请求结果类型的NULL指针。（C++14之后）
* 否则，new表达式不会调用分配函数，取而代之的是，抛出类型异常std::bad\_array\_new\_length或从它派生的异常。（C++11之后）

第一维为0是可以接受的，且会调用分配函数。

**注意**：

std::vector 提供与一维动态数组相似的功能。

#### 4.4.10.3 分配

new 表达式通过调用合适的分配函数分配存储空间。当type是非数组类型时，调用函数**operator new**。当是数组类型时，调用函数**operator new[]**。

C++程序提供全局的且特定于类的分配函数的代替者。如果new表达式是以可选的::操作符开始的话，如::new T或::new T[n]，特定于类的替代函数就会被忽略（函数在全局范围内查找）。否则，如果T是一个class类型，查找在T的类的作用域内查找。

当调用分配函数时，new表达式要求第一个参数传递请求存储空间的大小，类型为std::size\_t，对于非数组类型T可以通过sizeof(T)计算。

数组分配时可以在为数组分配的存储空间的前面提供额外的存储空间，随着对new的调用的改变而变化。new表达式返回的指针在内存分配函数返回的指针发生偏移之后的结果。许多实现使用数组的额外的存储空间来存储数组中的对象数量，其被delete[]表达式使用，调用正确个数的析构函数。另外，如果使用new创建char，unsigned char，或std::byte类型的数组时，如果需要保证所有不大于所请求数组大小的对象类型的对齐方式正确，则可能从分配函数申请额外的存储空间。

|  |  |
| --- | --- |
| New-expressions are allowed to elide or combine allocations made through replaceable allocation functions. In case of elision, the storage may be provided by the compiler without making the call to an allocation function (this also permits optimizing out unused new-expression). In case of combining, the allocation made by a new-expression E1 may be extended to provide additional storage for another new-expression E2 if all of the following is true:   1. The lifetime of the object allocated by E1 strictly contains the lifetime of the object allocated by E2, 2. E1 and E2 would invoke the same replaceable global allocation function 3. For a throwing allocation function, exceptions in E1 and E2 would be first caught in the same handler.   Note that this optimization is only permitted when new-expressions are used, not any other methods to call a replaceable allocation function: delete[] new int[10]; can be optimized out, but operator delete(operator new(10)); cannot. | C++14之后 |

#### 4.4.10.4 Placement new

如果placement\_params被提供，它们会作为额外参数传递给分配函数。这类分配函数就是“placement new”，定义如下，

|  |
| --- |
| void\* operator new(std::size\_t, void\*) |

返回未作更改的第2个参数。这被用于在已分配的存储空间上构造对象：

|  |
| --- |
| char\* ptr = new char[sizeof(T)]; // 分配内存  T\* tptr = new(ptr) T; // 在已分配存储（“位置”）构造  tptr->~T(); // 析构  delete[] ptr; // 解分配内存 |

注意：分配器 (Allocator) 类的成员函数封装此功能。

|  |  |
| --- | --- |
| 当分配的对象的对齐要求超过\_\_STDCPP\_DEFAULT\_NEW\_ALIGNMENT\_\_或者这类对象的数组时，new表达式会传递对齐要求（包含在std::align\_val\_t中）给分配函数，作为第2参数（对于放置形式，placement\_params出现在对齐要求之后，作为第3个，第4个等参数）。如果重载失败（当特定于类的分配函数被使用不同的签名定义时会发生，因为它隐藏了全局性），会在参数列表中没有对齐要求的情况下尝试第2次重载解析。这允许对齐不敏感的特定于类的分配函数优先于全局的、对齐敏感的分配函数。 | C++17之后 |

|  |
| --- |
| new T; // 调用operator new(sizeof(T))  // (C++17)或operator new(sizeof(T), std::align\_val\_t(alignof(T))))  new T[5]; // 调用operator new[](sizeof(T)\*5 + overhead)  // (C++17) 或operator new(sizeof(T)\*5+overhead, std::align\_val\_t(alignof(T))))  new(2,f) T; // 调用operator new(sizeof(T), 2, f)  // (C++17) 或operator new(sizeof(T), std::align\_val\_t(alignof(T)), 2, f) |

If the allocation function return a null pointer, which is possible if the non-throwing overload was selected, e.g. with new(std::nothrow) T;, then the new-expression returns immediately, it does not attempt to initialize an object or to call a deallocation function. If the standard placement allocation function returns a null pointer, which is possible if the user passes a null pointer as the argument, the behavior is undefined. (since C++17)

#### 4.4.10.4 构造函数

由new表达式创建的对象的初始化遵循下面的规则：

* 对于非数组类型，在已分配的内存上构造对象。
  + 如果初始化器不存在，则每一个元素被[默认初始化](#_6.1_默认初始化)。
  + 如果初始化器是用括号括起来的参数列表，对象被[直接初始化](#_6.4_Direct_initialization)。
  + 如果初始化器是用大括号括起来的参数列表，对象被[列表初始化](#_6.6_列表初始化(C++11))。（C++11之后）
* 对于数组类型，对象数组被初始化。
  + 如果初始化器不存在，则每一个元素被[默认初始化](#_6.1_默认初始化)。
  + 如果初始化器是一对空括号，每一个元素被[值初始化](#_6.2_值初始化(C++03))。
  + 如果初始化器是用大括号括起来的参数列表，数组被[聚合初始化](#_6.5_Aggregate_initialization)。（C++11之后）

如果初始化引抛出异常而终止（例如，在构造函数里），如果new表达式已经分配了内存，则会调用合适的析构函数，释放内存：对于非数组类型调用operator delete，数组类型调用operator delete[]。如果new表达式使用了::new语法，则在全局作用域内查找析构函数，否则，如果T是一个class类型的话，就在T的作用域范围内查找。如果失败的内存分配函数是普通的（非placement），则对析构函数的查找遵循delete表达式所描述的规则。对于失败的placement new，除了第一个参数之外，将要调用的析构函数的所有的参数必须和placement new的参数相同。将先前从分配函数取得的值作为第一参数，对齐方式作为可选对齐参数（C++17之后），如果有placement\_params，作为额外的placement参数。如果没有发现析构函数，内存不会被释放。

#### 4.4.10.5 内存泄漏

由new表达式创建的对象（具有动态存储周期）一直存在，直到使用配对的delete表达式删除这个对象。如果存储空间的原指针丢失，则对象变得不可用，也不可析构：内存泄漏就会发生。

1. 指针赋空值：

|  |
| --- |
| int\* p = new int(7); // 动态分配int对象，初始值为7  p = nullptr; // 内存泄漏 |

2. 指针离开作用域：

|  |
| --- |
| void f()  {  int\* p = new int(7);  } // 内存泄漏 |

3. 因为异常：

|  |
| --- |
| void f()  {  int\* p = new int(7);  g(); // 可能会异常抛出  delete p; // 如果没有异常，就会正常  } // 如果g()发生异常抛出，就会内存泄漏 |

为了简化动态内存的对象管理，通常将new表达式的结果存储于智能指针：std::auto\_ptr（C++17之前）std::unique\_ptr，或者std::shared\_ptr（C++11之后）。这些指针保证了在上述情况下执行delete表达式。

### 4.4.11 delete

销毁由[new表达式](#_4.4.10_new)创建的对象，并释放获得的内存区域。

#### 4.4.11.1 语法

|  |  |
| --- | --- |
| ::(可选) delete *expression* | （1） |
| ::(可选) delete [] *expression* | （2） |

1. 销毁由[new表达式](#_4.4.10_new)创建的非数组对象
2. 销毁由[new[]表达式](#_4.4.10_new)创建的数组对象

#### 4.4.11.2 解释

对于第1种形式（非数组），*expression* 必须是对象类型的指针或上下文可隐式转换成这类指针的class类型指针，它的值可以是空指针或指向new创建的非数组对象的指针，或者非数组对象的基对象指针。如果表达式*expression* 是其它的，比如new（数组形式）创建的指针，行为未定义。

第2种形式（数组），*expression*必须是new[]获得的指针值或空指针。如果表达式*expression*是其它的，比如new（非数组形式）创建的指针，行为未定义。

该表达式返回void类型。

如果被删除对象在删除的时间点上具有不完整的class类型，且具有非凡的析构函数（non-trivial destructor）或释放内存的函数的完整类，行为未定义。

如果expression不是空指针，且内存释放函数不是正在销毁的delete（C++20之后），delete表达式调用正在被销毁的对象，或者正在被销毁的数组的每一个元素的析构函数（如果有的话）。

之后，无论析构函数是否抛出异常，delete表达式都会调用内存释放函数：operator delete或operator delete[]，除非配对的new表达式和另一个new表达式组合使用（C++14之后）。

The deallocation function's name is looked up in the scope of the dynamic type of the object pointed to by expression, which means class-specific deallocation functions, if present, are found before the global ones. If :: is present in the delete expression, only the global namespace is examined by this lookup.

If lookup finds more than one deallocation function, the function to be called is selected as follows (see deallocation function for a more detailed description of these functions and their effects):

|  |  |
| --- | --- |
| * If at least one of the deallocation functions is a destroying delete, all non-destroying deletes are ignored. | （C++20之后） |
| * If the type's alignment requirement exceeds \_\_STDCPP\_DEFAULT\_NEW\_ALIGNMENT\_\_, alignment-aware deallocation functions (with a parameter of type std::align\_val\_t) are preferred. For other types, the alignment-unaware deallocation functions (without a parameter of type std::align\_val\_t) are preferred.   + If more than one preferred functions are found, only preferred functions are considered in the next step.   + If no preferred functions are found, the non-preferred ones are considered in the next step. * If only one function is left, that function is selected. | （C++17之后） |

* If the deallocation functions that were found are class-specific, size-unaware class-specific deallocation function (without a parameter of type std::size\_t) is preferred over size-aware class-specific deallocation function (with a parameter of type std::size\_t)

|  |  |
| --- | --- |
| * Otherwise, lookup reached global scope, and: * If the type is complete and if, for the delete[] only, the operand is a pointer to a class type with a non-trivial destructor or a (possibly multi-dimensional) array thereof, the global size-aware global function (with a parameter of type std::size\_t) is selected * Otherwise, it is unspecified whether the global size-aware deallocation function (with a parameter of type std::size\_t) or the global size-unaware deallocation function (without a parameter of type std::size\_t) is selected | C++14之后 |

The pointer to the block of storage to be reclaimed is passed to the deallocation function that was selected by the process above as the first argument. The size of the block is passed as the optional std::size\_t argument. The alignment requirement is passed as the optional std::align\_val\_t argument. (since C++17)

|  |  |
| --- | --- |
| If expression evaluates to a null pointer value, no destructors are called, and the deallocation function may or may not be called (it's implementation-defined), but the default deallocation functions are guaranteed to do nothing when handed a null pointer. | C++14之前 |
| If expression evaluates to a null pointer value, no destructors are called, and the deallocation function is not called. | C++14之后 |

If expression evaluates to a pointer to a base class subobject of the object that was allocated with new, the destructor of the base class must be virtual, otherwise the behavior is undefined.

#### 4.4.11.3 注意

A pointer to void cannot be deleted because it is not a pointer to a complete object type.

|  |  |  |
| --- | --- | --- |
| Because a pair of brackets following the keyword delete is always interpreted as the array form of delete, a lambda-expression with empty capture list immediately after delete must be enclosed in parentheses.   |  | | --- | | // delete []{return new int; }(); // parse error  delete ([]{return new int; })(); // OK | | C++11之后 |

### 4.4.12 typeid

查询类型信息。在必须知道多态对象的动态类型和用于静态类型标识符的情况下使用。

#### 4.4.12.1 语法及解释

1 语法

typeid (类型或表达式)

当使用该操作符时，必须包含头文件<typeinfo>。

typeid表达式是一个左值表达式，它引用一个具有静态存储生命周期的对象，其类型是由语句const std::type\_info指定的多态类型或者派生于它的类型。

2 解释

1. 其返回类型是引用std::type\_info对象的引用类型；
2. 检查表达式，
   1. 如果表达式是泛左值表达式，标识了一个多态类型的对象（也就是说，类声明或者继承了至少一个虚函数），typeid表达式求值然后引用std::type\_info对象表示该表达式的多态类型。如果泛左值表达式被指针操作符\*且该指针是null指针，类型std::bad\_typeid或派生于它的异常就会被抛出。
   2. 如果表达式不是多态类型的泛左值表达式，typeid不会求值。

在所有的情况中，typeid会忽略cv限定符（也就是说，typeid(T）== typeid(const T))。

如果typeid的算子是class类型或class的引用类型，class类型绝不能是[不完整类型](#_1.4.6_不完整类型)。

如果typeid被用在正在构建或销毁的对象（析构或者构造函数里，包括构造函数的初始化列表或默认成员初始化），由typeid引用的std::type\_info对象就会表示正在构造或者销毁的类，即使它不是最底层的类。

#### 4.4.12.2 注意事项

当应用于多态类型表达式时，对typeid的求值可能涉及运行时开销（虚表查找），否则typeid表达式将在编译时解析。

在程序结束是，没有指定typeid引用对象的析构函数是否被调用。

对于相同的类型，使用typeid表达式并求值，但是所返回的引用std::type\_info实例并不能保证相同；尽管它们的std::type\_index和std::type\_info::hash\_code是相同的。

|  |
| --- |
| const std::type\_info& ti1 = typeid(A);  const std::type\_info& ti2 = typeid(A);    assert(&ti1 == &ti2); // 不保证  assert(ti1.hash\_code() == ti2.hash\_code()); // 保证  assert(std::type\_index(ti1) == std::type\_index(ti2)); // 保证 |

#### 4.4.12.3 举例说明

该示例使用type\_info::name返回完整类型名称的实现之一显示输出; 通过c++ filt -t筛选，如果使用gcc或类似的。

|  |
| --- |
| #include <iostream>  #include <string>  #include <typeinfo>    struct Base {}; // 非多态  struct Derived : Base {};    struct Base2 { virtual void foo() {} }; // 多态  struct Derived2 : Base2 {};    int main() {  int myint = 50;  std::string mystr = "string";  double\* mydoubleptr = nullptr;    std::cout << "myint has type: " << typeid(myint).name() << '\n'  << "mystr has type: " << typeid(mystr).name() << '\n'  << "mydoubleptr has type: " << typeid(mydoubleptr).name() << '\n';    // std::cout << myint是多态类型的泛左值表达式；它会被求值  const std::type\_info& r1 = typeid(std::cout << myint);  std::cout << '\n' << "std::cout<<myint has type : " << r1.name() << '\n';    // std::printf()不是多态类型的泛左值表达式；不会被求值  const std::type\_info& r2 = typeid(std::printf("%d\n", myint));  std::cout << "printf(\"%d\\n\",myint) has type : " << r2.name() << '\n';    // 非多态左值是一个static类型  Derived d1;  Base& b1 = d1;  std::cout << "reference to non-polymorphic base: " << typeid(b1).name() << '\n';    Derived2 d2;  Base2& b2 = d2;  std::cout << "reference to polymorphic base: " << typeid(b2).name() << '\n';    try {  // 解引用一个null指针： 对于非多态表达式是OK的  std::cout << "mydoubleptr points to " << typeid(\*mydoubleptr).name() << '\n';  // 解引用一个null指针： 对于多态表达式不是OK的  Derived2\* bad\_ptr = nullptr;  std::cout << "bad\_ptr points to... ";  std::cout << typeid(\*bad\_ptr).name() << '\n';  } catch (const std::bad\_typeid& e) {  std::cout << " caught " << e.what() << '\n';  }  } |

可能的输出结果：

|  |
| --- |
| myint has type: i  mystr has type: NSt7\_\_cxx1112basic\_stringIcSt11char\_traitsIcESaIcEEE  mydoubleptr has type: Pd  5050  std::cout<<myint has type : So  printf("%d\n",myint) has type : i  reference to non-polymorphic base: 4Base  reference to polymorphic base: 8Derived2  mydoubleptr points to d  bad\_ptr points to... caught std::bad\_typeid |

## 4.5 操作符重载（unedit）

## 4.6 默认比较(C++20) （unedit）

## 4.7 运算符优先级（unedit）

## 4.8 转换（incomplete）

### 4.8.1 隐式转换

当在上下文中不接受某一类型为T1的表达式而接受的类型为T2时，就会发生T1到T2的隐式转换，尤其是：

* 当调用参数类型为T2，但是传入的实参类型为T1的表达式时；
* 当操作符期望的类型为T2，但是算子类型为T1的表达式时；
* 当初始化一个类型为T2的新对象时，包含返回类型为T2的函数返回语句；
* 当在switch语句中使用T1表达式时（T2是整数类型）；
* 当在if语句或者循环语句中使用该表达式时（T2是bool型）；

只有存在一个从T1到T2的没有歧义的隐式转换序列时，该程序才能符合语法规则（能够被编译）。

如果调用的函数或运算符有多个重载，则在从T1到每个可用的T2构建隐式转换序列后，[重载决策](#_7.7_重载决策)规则将决定编译哪个重载版本。

注意：在算术表达式中，对二元操作符的操作算子作隐式转换，其目标类型是由一组单独的规则决定的，称之为通用算术转换。

#### 4.8.1.1 转换顺序

隐式转换序列按以下顺序组成：

1) 零或一个标准转换序列；

2) 零或一个用户自定义转换序列；

3) 零或一个标准转换序列。

在考虑构造函数或用户定义的转换函数的参数时，只允许一个标准转换序列（否则用户定义的转换可以有效地链接）。 从一种内置类型转换为另一种内置类型时，只允许使用一种标准转换序列。

标准转换序列按以下顺序组成：

* 0或左值变换；
* 零或一个数字提升或数字转换;
* 零或一个函数指针转换;（C++17之后）
* 零或一个资格调整。

用户定义的转换由零个或一个非显式单参数构造函数或非显式转换函数调用组成

当且仅当T2可以从e被复制初始化时，表达式e被认为是隐式转换为T2的，即声明T2 t = e;是格式良好的（可以编译），对于一些发明的临时t。请注意，这与直接初始化（T2 t（e））不同，其中将额外考虑显式构造函数和转换函数。

#### 4.8.1.2 上下文转换

在下面的五个上下文中，期望的类型是bool型，如果声明语句bool t(e);符合语法规则，则隐式转换被自动执行。也就是说，诸如用户自定义的显式转换函数T :: operator bool（）const;也会被考虑。这类表达式e被称为上下文可转换为bool型的表达式。（C++11之后）

* 条件控制语句if，while，for；
* 逻辑操作符!，&&和||；
* 条件操作符?:；
* static\_assert语句；
* noexcept。

In the following contexts, a context-specific type T is expected, and the expression e of class type E is only allowed if E has a single non-explicit user-defined conversion function to an allowable type (until C++14)there is exactly one type T among the allowable types such that E has non-explicit conversion functions whose return types are (possibly cv-qualified) T or reference to (possibly cv-qualified) T, and e is implicitly convertible to T (since C++14). Such expression e is said to be contextually implicitly converted to the specified type T. Note that explicit conversion functions are not considered, even though they are considered in contextual conversions to bool. (since C++11)

在以下上下文中，期望特定于上下文的类型T，并且仅当E具有单个非显式用户定义的转换函数到允许类型（直到C ++ 14）时才允许类类型E的表达式e 在允许的类型中恰好是一个类型T，使得E具有非显式转换函数，其返回类型是（可能是cv-qualified）T或对（可能是cv-qualified）T的引用，并且e可以隐式转换为T（因为C++14）。 据说这种表达式e被上下文隐式地转换为指定的类型T.注意，不考虑显式转换函数，尽管它们被认为是对bool的上下文转换。 （自C ++ 11以来）

* [delete表达式](#_4.4.9_delete)的参数（T是任何对象指针类型）；
* 整型常量表达式，其中使用了一个文本类（T是任何整型或未限定作用域的枚举类型，所选用户定义的转换函数必须是constexpr）；
* switch语句的控制表达式（T是任何整数或枚举类型）。

|  |
| --- |
| #include <cassert>    template<typename T>  class zero\_init  {  T val;  public:  zero\_init() : val(static\_cast<T>(0)) { }  zero\_init(T val) : val(val) { }  operator T&() { return val; }  operator T() const { return val; }  };    int main()  {  zero\_init<int> i; assert(i == 0);  i = 7; assert(i == 7);  switch(i) { } // error until C++14 (more than one conversion function)  // OK since C++14 (both functions convert to the same type int)  switch(i + 0) { } // always okay (implicit conversion)  } |

#### 4.8.1.3 值转换

值转换是改变表达式的[值类别](#_4.1_值类别)的转换。只要表达式作为一个操作符的操作数出现，就会发生这种情况，该操作符期望表达不同的值类别。

**1. 左值到右值的转换**

A glvalue of any non-function, non-array type T can be implicitly converted to a prvalue of the same type. If T is a non-class type, this conversion also removes cv-qualifiers. If the glvalue has the type std::nullptr\_t, the resulting prvalue is the null pointer constant nullptr.

Unless encountered in unevaluated context (in an operand of sizeof, typeid, noexcept, or decltype), this conversion effectively copy-constructs a temporary object of type T using the original glvalue as the constructor argument, and that temporary object is returned as a prvalue.

This conversion models the act of reading a value from a memory location into a CPU register.

If the object to which the glvalue refers contains an indeterminate value (such as obtained by default initializing a non-class automatic variable), the behavior is undefined

|  |  |
| --- | --- |
| except if the indeterminate value is of possibly cv-qualified unsigned character type which was not cached in a CPU register, or, formally:   * its storage duration was static or thread-local; * or a pointer to it was constructed; * or it was bound to a reference.   The behavior is also implementation-defined (rather than undefined) if the glvalue contains a pointer value that was invalidated by delete. | C++11之后 |

**2. 数组向指针的转换**

An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be implicitly converted to a prvalue of type "pointer to T". If the array is a prvalue, temporary materialization occurs. (since C++17) The resulting pointer refers to the first element of the array (see array to pointer decay for details)

**3. 临时物化（C++17之后）**

任何完整类型T的纯右值都可以转换为相同类型T的消亡值。该转换对纯右值进行求值，然后初始化一个相同类型T的临时对象，并把它作为结果对象，产生一个消亡值表示这个临时对象。如果T是类或类类型的数组，它必须具有一个可访问的且不可删除的析构函数。

|  |
| --- |
| struct S { int m; };  int i = S().m; // 从C++17开始，成员访问期待泛左值；  // S() 这个纯右值被转换为消亡值 |

临时实体化发生在下列情况下：

1. 当绑定引用到纯右值时；
2. 当对一个类纯右值执行成员访问时；
3. 当执行数组向指针的转化（见上面）或下标访问一个数组纯右值；
4. 当初始化std::initializer\_list<T>类型的对象；
5. 当typeid被应用到纯右值上时(这是非求值表达式的一部分);
6. 当sizeof应用到纯右值上(这是非求值表达式的一部分)；
7. 当纯右值表现为一个弃值表达式时。

考虑如下代码：

|  |
| --- |
| struct S  {  public:  S(int i) { m = i; }  ~S() { std::cout << " ~S() called \n"; }  int m;  };  void fun(S s)  {  std::cout << "fun called \n";  }  int main()  {  fun(S(1));  } |

S(1)构造了一次，是个未命名的临时对象（暂且给它取名为temp）。然后传递给函数fun()，s是另一个局部对象。因此，程序应该打印两次“~S() called”。（实测vs2015打印2次，gcc和clang中只打印一次）

从C++17开始，S(1)这个纯右值不一定要物化为temp临时变量，意思是在这个程序中，根本就没有创建temp临时对象。因此，在C++17支持的编译器中，只打印一次“~S() called”。

这绝非是编译器厂家自由选择的优化技术，编译器的优化技术认为，S(1)和fun(S s)中的s是一回事，就直接在s的构造上做文章。（见《Inside C++ Model》）

C++强制要求纯右值只是初始化语义，也就是说，纯右值只能帮助其他对象初始化，自己是不能轻易物化的。这里，fun(S(1))就没有发生临时物化。那么，何时会发生呢？例如：

int k = S().m;

这样子，S()返回的纯右值就必须进行临时物化，因为要访问它的对象m。

**4. 函数向指针的转换**

函数类型T的左值可以隐式转换为指向该函数的纯右值指针。这不适用于非静态成员函数，因为指向非静态成员函数的左值不存在。

#### 4.8.1.4 数值提升

**1. 整数提升**

**2. 浮点数提升**

类型为float的纯右值能够被转换为类型double的纯右值。数值不会发生变化。

#### 4.8.1.5 数值转换

1. 整数转换

2. 浮点数转换

3. 浮点数向整数转换

4. 指针转换

5. 指针向成员转换

6. bool转换

7 限定性转换

8 函数指针转换

#### 4.8.1.6 安全bool问题

### 4.8.2 显式转换

### 4.8.3 用户自定义转换

保证一个class类型向另一个类型的隐式转换或显式转换。

#### 4.8.3.1 语法与解释

转换函数被声明为非static成员函数或不带参数的成员函数模板，并且还得是非显式返回类型，

|  |  |  |
| --- | --- | --- |
| operator conversion-type-id | （1） |  |
| explicit operator conversion-type-id | （2） | C++11 |

1. 声明一个用户自定义的转换函数，可以被所有的隐式或显式转换调用
2. 声明一个用户自定义的转换函数，只能被直接初始化和显式转换调用

conversion-type-id 就是一个类型ID（type-id），例外的情况就是函数和数组操作符[]或()，不允许出现在它的声明符中（因而，往这些类型的转换，比如指针向数组的转换就要求使用类型别名alias/typedef或ID模板：具体的可以参考下面的例子）。

注意：也就是说，不管有没有typedef，conversion-type-id都不能表示数组或函数类型

#### 4.8.3.2 示例

|  |
| --- |
| struct X {  // 隐式转换-X对象向int转换的时候，都会被转换成7  operator int() const { return 7; }    // 显式转换-所有的int型指针都返回nullptr  explicit operator int\*() const { return nullptr; }    // 错误: 数组操作符不允许出现在conversion-type-id中  // operator int(\*)[3]() const { return nullptr; }  using arr\_t = int[3];  operator arr\_t\*() const { return nullptr; } // 通过类型说明符typedef是可以的  // operator arr\_t () const; // 错误: 转换成数组在任何情况下都是不被允许的。  };    int main()  {  X x;    int n = static\_cast<int>(x); // OK: sets n to 7  int m = x; // OK: sets m to 7    int\* p = static\_cast<int\*>(x); // OK: sets p to null  // int\* q = x; // Error: 这不是一个隐式转换，所以不能使用拷贝方法    int (\*pa)[3] = x; // OK  } |

### 4.8.4 static\_cast

使用隐式转换，或者用户自定义的转换的组合，进行类型转换。

#### 4.8.4.1 语法

static\_cast <*新类型*> (*表达式*)，返回值就是新类型的值。

#### 4.8.4.2 解释

只有下面这些转换能够使用static\_cast完成，例外的情况就是，当转换丢掉常量性或易失性时。

1. 如果存在从表达式到新类型的[隐式转换](#_4.8.1_隐式转换)，或者，如果对象的直接初始化或来自表达式的新类型的引用的重载策略发现至少有一个可执行函数，则static\_cast将产生一个虚构的变量Temp，它将会如同下面的语句一样进行初始化：  
    新类型 Temp（表达式）  
   调用新类型的构造函数或者用户自定义的转换操作符，会涉及隐式转换。
2. 如果新类型是某一类结构D的指针或引用，表达式的类型是D的非虚基类B的指针或引用时，static\_cast执行downcast。如果B是歧义的，无法访问的，或者是D的虚基类，那么这个downcast就是语法错误的。这样的static\_cast不进行运行时检查以确保对象的运行时类型是真正的D，所以，只有在通过其它方法保证这个假设条件成立时，才能安全地使用static\_cast。比如说，实现静态多态。所以，安全的downcast可以使用dynamic\_cast完成。
3. 如果新类型是一个右值引用类型，static\_cast把泛左值，类纯右值或数组纯右值和左值表达式转换成消亡值。

3) If new\_type is an rvalue reference type, static\_cast converts the value of glvalue, class prvalue, or array prvalue (until C++17)any lvalue (since C++17) expression to xvalue referring to the same object as the expression, or to its base sub-object (depending on new\_type). If the target type is an inaccessible or ambiguous base of the type of the expression, the program is ill-formed. If the expression is a bit field lvalue, it is first converted to prvalue of the underlying type. This type of static\_cast is used to implement move semantics in std::move.

(since C++11)

7) Scoped enumeration type can be converted to an integer or floating-point type. When the target type is cv bool, the result is false if the original value is zero and true for all other values. For the remaining integral types, the result is the value of the enum if it can be represented by the target type and unspecified otherwise. (since C++11)

8) A value of integer or enumeration type can be converted to any complete enumeration type.

If the underlying type is not fixed, the result is unspecified (until C++17)undefined behavior (since C++17) if the value of expression is out of range (the range is all values possible for the smallest bit field large enough to hold all enumerators of the target enumeration).

If the underlying type is fixed, the result is the same as the converting the original value first to the underlying type of the enumeration and then to the enumeration type.

A value of a floating-point type can also be converted to any complete enumeration type.

The result is the same as converting the original value first to the underlying type of the enumeration, and then to the enumeration type.

9) A pointer to member of some class D can be upcast to a pointer to member of its unambiguous, accessible base class B. This static\_cast makes no checks to ensure the member actually exists in the runtime type of the pointed-to object.

10) A prvalue of type pointer to void (possibly cv-qualified) can be converted to pointer to any object type. If the original pointer value represents an address of a byte in memory that does not satisfy the alignment requirement of the target type, then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object a, and there is an object b of the target type (ignoring cv-qualification) that is pointer-interconvertible (as defined below) with a, the result is a pointer to b. Otherwise the pointer value is unchanged. Conversion of any pointer to pointer to void and back to pointer to the original (or more cv-qualified) type preserves its original value.

3）如果new\_type是右值引用类型，则static\_cast将glvalue，class prvalue或array prvalue的值（直到C ++ 17）转换为与表达式相同的对象的xvalue或者其基础子对象（取决于new\_type）。如果目标类型是表达式类型的不可访问或不明确的基础，则该程序是格式不正确的。如果表达式是位域左值，则它首先转换为基础类型的前值。这种类型的static\_cast用于在std :: move中实现移动语义。（自C ++ 11以来）

4）如果new\_type是void类型（可能是cv-qualified），则static\_cast在求值后丢弃表达式的值。

5）如果存在从新类型到表达式类型的标准转换序列，其不包括左值到右值，数组到指针，函数到指针，空指针，空成员指针，函数指针（C++17之后）或bool转换，然后static\_cast可以执行该隐式转换的逆操作。

6）如果将表达式转换为新类型涉及左值到右值，数组到指针或函数到指针的转换，则可以通过static\_cast显式执行。

7）作用域枚举类型可以转换为整数或浮点类型。当目标类型为cv布尔型时，如果原始值为零，则结果为false，对于所有其他值，则结果为true。对于其余的整型，结果是枚举的值，如果它可以由目标类型表示，否则不指定。 （C++11之后）

8）整数或枚举类型的值可以转换为任何完整的枚举类型。

* 如果基础类型不固定，则结果是未指定的（C++17之前）未定义的行为（C++17以后）如果表达式的值超出范围（范围是所有可能的最小位域的值足以容纳目标枚举的所有枚举数）。
* 如果基础类型是固定的，则结果与将原始值首先转换为枚举的基础类型然后转换为枚举类型的结果相同。
* 浮点类型的值也可以转换为任何完整的枚举类型。
* 结果与将原始值首先转换为枚举的基础类型，然后转换为枚举类型的结果相同。

9）指向某个D类成员的指针可以被上传到指向其明确可访问基类B的成员的指针。此static\_cast不进行检查，以确保该成员实际上存在于指向对象的运行时类型中。

10）指向void的指针（可能是cv-qualified）的指针可以转换为指向任何对象类型的指针。如果原始指针值代表内存中不满足目标类型对齐要求的字节地址，则未指定结果指针值。否则，如果原始指针值指向一个对象a，并且有一个目标类型的对象b（忽略cv-qualification），该对象b是指针互换（如下定义），则结果是一个指向b的指针。否则指针值不变。任何指针指向void的指针转换回指向原始（或更多cv-qualified）类型的指针都会保留其原始值。

与所有转换表达式一样，结果是：

* 如果新类型是左值引用类型或对函数类型的右值引用，则为左值;
* 如果新类型是对对象类型的右值引用，则为消亡值;
* 否则就是一个纯右值。

如果满足以下条件，则两个对象a和b是指针互换的：

* 它们是同一个对象，或者
* 一个是union对象，另一个是该对象的非静态数据成员，或者
* 一个是标准布局类对象，另一个是该对象的第一个非静态数据成员，或者如果该对象没有非静态数据成员，则该对象的第一个基类子对象或
* 存在一个对象c，使得a和c是指针可互换的，而c和b是指针可互换的。

#### 4.8.4.3 举例

|  |
| --- |
| #include <vector>  #include <iostream>  struct B {  int m = 0;  void hello() const {  std::cout << "Hello world, this is B!\n";  }  };  struct D : B {  void hello() const {  std::cout << "Hello world, this is D!\n";  }  };  enum class E { ONE = 1, TWO, THREE };  enum EU { ONE = 1, TWO, THREE };  int main()  {  // 1: 初始化转换  int n = static\_cast<int>(3.14);  std::cout << "n = " << n << '\n';  std::vector<int> v = static\_cast<std::vector<int>>(10);  std::cout << "v.size() = " << v.size() << '\n';    // 2: static 向下转换  D d;  B& br = d; // 通过隐式转换向上转换  br.hello();  D& another\_d = static\_cast<D&>(br); // downcast  another\_d.hello();    // 3: lvalue to xvalue  std::vector<int> v2 = static\_cast<std::vector<int>&&>(v);  std::cout << "after move, v.size() = " << v.size() << '\n';    // 4: discarded-value expression  static\_cast<void>(v2.size());    // 5. inverse of implicit conversion  void\* nv = &n;  int\* ni = static\_cast<int\*>(nv);  std::cout << "\*ni = " << \*ni << '\n';    // 6. array-to-pointer followed by upcast  D a[10];  B\* dp = static\_cast<B\*>(a);    // 7. scoped enum to int or float  E e = E::ONE;  int one = static\_cast<int>(e);  std::cout << one << '\n';    // 8. int to enum, enum to another enum  E e2 = static\_cast<E>(one);  EU eu = static\_cast<EU>(e2);    // 9. pointer to member upcast  int D::\*pm = &D::m;  std::cout << br.\*static\_cast<int B::\*>(pm) << '\n';    // 10. void\* to any type  void\* voidp = &e;  std::vector<int>\* p = static\_cast<std::vector<int>\*>(voidp);  } |

### 4.8.5 dynamic\_cast

安全地将对class的指针和引用，

#### 4.8.5.1 语法与解释

语法：dynamic\_cast<**new\_type**>(**expression**)

1. new\_type: 指向完整class类型的指针或引用，也可以是指向void的指针
2. expression：如果new\_type是引用，则expression应该是完整class类型的左值（C++11之后扩展到泛左值），而如果是指针，则应该是指向完整class类型的纯右值。

如果强制转换成功，则dynamic\_cast返回new\_type类型的值。如果转换失败，且new\_type是一个指针类型，则返回该类型的null指针。如果转换失败，且new\_type是一个引用类型，则抛出异常，调用匹配std::bad\_cast类型的处理程序。

下面的情况可以使用dynamic\_cast，但是如果转换会抛弃常量性（constness）或易变性（volatility），则不可以使用dynamic\_cast。

1. 如果要转换的expression完全就是新的类型或者是新类型的没有进行cv限定的版本的话，就可以使用dynamic\_cast（换句话说，就是dynamic\_cast可以用来添加常量性，隐式转换和static\_cast也可以实现这种转换）。

|  |
| --- |
| using namespace std;  class Fuck  {  public:  int a;  Fuck() : a(10) {}  ~Fuck() {}  };  int main()  {  Fuck f;  const Fuck\* fuck = dynamic\_cast<const Fuck\* >(&f);// 添加常量性，ok  const Fuck fuck\_b = dynamic\_cast<const Fuck>(f);// 错误，必须是指向完整class类型的指针和引用  const Fuck cf;  Fuck\* pc = dynamic\_cast<Fuck\*>(&cf);// 删除常量性，这个工作应该有const\_cast完成  cout << fuck->a << endl;  } |

1. 空指针转换为新类型的null指针。（这种情况很好理解）
2. 如果expression的值是指向基类的指针或引用，或expression值是指向派生类的指针或引用，在这儿，基类是唯一的，是派生类可访问的基类，转换的结果就是表达式指向的派生类对象里的基类子对象的指针或引用。（通俗地讲，就是把类指针或引用向上转换。隐式转换和static\_cast也可以实现这种转换，详细细节可以参考下面代码中关于“向上转换”的实例）。
3. 如果expression是指向多态类型的对象的指针的话，而新类型是指向void的指针，那么转换的结果就是指向最底层派生类的指针。
4. 如果expression是指向多态类型的基类的指针或引用，而新类型是指向派生类的指针或引用（会进行run-time检查）：
   1. 向下转换（downcast）- 通过expression可以找到该指针指向的最底层的派生对象（在继承层次中）。如果在最底层的派生对象中，expression指向或引用一个公共的基类的派生对象，且只要一个派生类型的子对象是派生自expression指定或引用的子对象。如下面的代码中，在继承层次中最底层的对象是d，expression在这里就是一个类型为struct A的引用，D的子对象就是A的对象本身。从基类指针向派生类指针转换，称之为“向下转换”。
   2. 侧向转换（sidecast）-如果最底层的派生类具有多个公共基类，而expression指向派生类对象的一个公共基类对象，而派生类具有清晰的公共基类的派生关系。具体看下面的代码演示。
   3. 运行时检查失败。如果dynamic\_cast被用在指针上，返回新类型的null指针。如果是引用，抛出std::bad\_cast的异常。

对于转换结果：

1. 如果新类型是左值引用类型，则转换结果就是左值（表达式必须是左值）
2. 如果新类型是右值引用类型，结果就是消亡值
3. 如果新类型为指针类型，则结果是纯右值。

#### 4.8.5.2 示例

|  |
| --- |
| #include <iostream>    struct V {  virtual void f() {}; // 必须是多态才能使用运行时检查的 dynamic\_cast  };  struct A : virtual V {};  struct B : virtual V {  B(V\* v, A\* a) {  // 在构造阶段进行类型转换（看下面的D的构造函数的调用）  dynamic\_cast<B\*>(v); // 定义明确的: v的类型是 V\*, V 是 B 的基类, 结果是 B\*  dynamic\_cast<B\*>(a); // 未定义行为：a有类型A\*，A也不是B的基类  }  };  struct D : A, B {  D() : B(static\_cast<A\*>(this), this) { }  };    struct Base {  virtual ~Base() {}  };    struct Derived: Base {  virtual void name() {}  };    int main()  {  D d; // 最底层的派生类对象  A& a = d; // 向上转换, 可以使用 dynamic\_cast，但不是必须的  D& new\_d = dynamic\_cast<D&>(a); // 向下转换  B& new\_b = dynamic\_cast<B&>(a); // 边缘转换-sidecast      Base\* b1 = new Base;  if(Derived\* d = dynamic\_cast<Derived\*>(b1))  {  std::cout << "downcast from b1 to d successful\n";  d->name(); // safe to call  }    Base\* b2 = new Derived;  if(Derived\* d = dynamic\_cast<Derived\*>(b2))  {  std::cout << "downcast from b2 to d successful\n";  d->name(); // safe to call  }    delete b1;  delete b2;  } |

### 4.8.6 const\_cast

### 4.8.7 reinterpret\_cast

## 4.9 文字常量

文字常量是C++程序源代码中嵌入的代表着常量值的符号。

（1）整数常量是整数类型的十进制，八进制，十六进制或二进制数字。

（2）字符常量是类型为char, char16\_t, char32\_t,或wchar\_t的单个字符。

（3）浮点型常量类型为float, double,或long double的值。

（4）字符串是类型为const char[], const char16\_t[], const char32\_t[], 或const wchar\_t[]的字符序列。

（5）bool型常量，true和false。

（6）nullptr，代表空指针，指针文字常量（C++11之后）。

（7）用户自定义文字常量，用户指定类型的常量值（C++11之后）。

### 4.9.1 boolean - integer - floating

### 4.9.2 character - string

### 4.9.3 nullptr(C++11)

### 4.9.4 user-defined (C++11)

# 5 声明

声明将名称引入（或重新引入）C++程序中。每种实体的声明都是不同的。定义是足以使用由名称标识的实体的声明。

声明是以下任一种：

（1）函数定义

（2）模板声明

（3）显式模板实例化

（4）显式模板特化

（5）[命名空间定义](#_5.1_命名空间)

（6）链接规范

（7）属性声明（attr；）（C++11之后）

（8）空声明（；）（C++11之后）

（9）没有decl-specifier-seq的函数声明

attr(optional) declarator ；

attr (since C++11) - sequence of any number of attributes

declarator - A function declarator.

This declaration must declare a constructor, destructor, or user-defined type conversion function. It can only be used as part of a template declaration, explicit specialization, or explicit instantiation.

（10）块声明（可以出现在块内的声明），这又可以是下列之一：

* asm定义
* 类型别名声明(since C++11)
* namespace alias definition
* using declaration
* using directive
* static\_assert declaration (since C++11)
* opaque enum declaration (since C++11)
* simple declaration

1. 简单声明

简单声明是引入，创建并且（可选地）初始化一个或多个标识符（通常是变量）的语句。

decl-specifier-seq init-declarator-list(optional) ； (1)

attr decl-specifier-seq init-declarator-list； (2)

attr（C++11之后） 任何数量的属性序列

decl-specifier-seq 说明符序列（见下文）。

init-declarator-list 用逗号（，）分割的、具有可选的初始化值的声明符列表。当声明一个命名的类、结构体、联合体、枚举时，init-declarator-list是可选的。

structured binding declaration也是简单声明。（C++17之后）

2. 说明符

声明说明符（decl-specifier-seq）是以空格分隔的下面的说明符的序列，任意顺序都可：

（1）typedef

（2）函数说明符

（3）inline

（4）friend

（5）constexpr

（6）存储类说明符（register，static，thread\_local(C++11之后），extern，mutable)

（7）类型说明符

|  |  |  |
| --- | --- | --- |
| class |  |  |
| enum |  |  |
| 简单类型说明符 | char, char16\_t, char32\_t (C++11之后), wchar\_t, bool, short, int, long, signed, unsigned, float, double, void |  |
|  | auto | C++11之后 |
|  | decltype说明符 |
|  | 先前声明的class名称（optionally qualified） |  |
|  | 先前声明的enum名称（optionally qualified） |  |
|  | 先前声明的typedef名称或类型别名(C++11之后) (optionally qualified) |  |

## 5.1 命名空间

命名空间提供给大项目一种避免命名冲突的方法。

在命名空间块中声明的符号被放置在一个命名范围内，以防止它们被误认为其它范围中的同名命名符号。

允许具有相同名称的多个命名空间块。这些块内的所有声明都在指定范围内被声明。

### 5.1.1语法

|  |  |  |
| --- | --- | --- |
| namespace ns\_name {declarations} | （1） |  |
| inline namespace ns\_name {declarations} | （2） | C++11之后 |
| namespace {declarations} | （3） |  |
| ns\_name：：name | （4） |  |
| using namespace ns\_name； | （5） |  |
| using ns\_name：：name； | （6） |  |
| namespace name = qualified-namespace ； | （7） |  |
| namespace ns\_name：：name | （8） | C++17之后 |

（1）为命名空间ns\_name指定定义。

（2）命名空间ns\_name的内联命名空间定义。 Declarations inside ns\_name will be visible in its enclosing namespace.

（3）未具名命名空间定义。它的成员从声明处到转译单元的结束处都有潜在地作用范围，具有内部链接属性。

注意：不知道名字当然无法访问了。不具名空间依然是外链接的，但是外界由于不知道名字所以无法访问，这样就具有了内链接的特性。使用不具名空间是为了保持对象的局部性。

可以用不具名命名空间替代static（staitc是内链接的）。

（4）命名空间名称（连同类名称）可以出现在作用域解析运算符的左侧，作为限定名称查找的一部分。

（5）using-directive： From the point of view of unqualified name lookup of any name after a using-directive and until the end of the scope in which it appears, every name from ns\_name is visible as if it were declared in the nearest enclosing namespace which contains both the using-directive and ns\_name.

（6）using-declaration： makes the symbol name from the namespace ns\_name accessible for unqualified lookup as if declared in the same class scope, block scope, or namespace as where this using-declaration appears.

（7）namespace-alias-definition： makes name a synonym for another namespace： see namespace alias

（8）nested namespace definition： **namespace A：：B：：C { ... }** is equivalent to **namespace A { namespace B { namespace C { ... } } }**.

### 5.1.2解释

#### 1 命名空间

|  |  |
| --- | --- |
| inline(optional) namespace attr(optional) identifier { namespace-body } | |
| inline | 如果存在，使其成为一个内联命名空间（参见下面）。如果原始名称空间定义不使用内联，则不能出现在扩展名称空间定义中。 |
| attr(C++17) | 任意数量的属性的可选序列 |
| identifier | 既可以是之前未使用的标识符，这种情况就是原始命名空间定义；也可以是命名空间的名称，在这种情况里，这是扩展命名空间定义；还可以是由：：分隔的封闭命名空间标识符的序列，以标识符结束，在这种情况下，就是嵌套命名空间定义（C++17后） |
| namespace-body | 可能是任何类型的声明序列（包括类和函数定义以及嵌套的命名空间） |

命名空间定义只能在命名空间范围内实现，包括全局范围。

为了重新打开一个已经存在的命名空间（形式上是一个扩展命名空间定义），在命名空间定义中使用的标识符的查找必须解析为一个命名空间名称（不是命名空间别名），该命名空间被声明为封闭命名空间的成员或封闭名称空间内的内联名称空间。

命名空间体定义了一个命名空间作用域，这会影响名称查找。

由命名空间体（包括嵌套名称空间定义）中出现的声明引入的所有名称都成为命名空间的成员，无论该命名空间定义是原始命名空间定义（引入的标识符）还是扩展命名空间定义（“重新打开”已经定义的命名空间）。

在命名空间体中声明的命名空间成员可以使用明确的限定条件在其外部定义或重新声明。

|  |
| --- |
| #include <algorithm>  #include <cctype>  #include <iostream>  #include <string>  #include <vector>  namespace Q {  namespace V { // V是Q的一个成员，它完全在Q内定义  // namespace Q：：V { // C++17 替换上面两行  // C是V的一个成员，它完全在V内定义；C：：m只被声明；  class C {  public：  void m()；  }；  // f是V的一个成员，在这里只定义  void f()；  }  // V外的V成员函数f的定义，f的封闭命名空间仍然是全局命名空间Q和Q：：V  void V：：f()  {  extern void h()； // This declares ：：Q：：V：：h  }  // 命名空间和类外 V：：C：：m 定义，封闭命名空间仍然是全局命名空间Q和Q：：V  void V：：C：：m()  {  printf("V：：C：：m()")；  }  }  namespace Q {  enum {  ABC = 'a'  }；  }  int main()  {  char abc[10] = {Q：：ABC}；  Q：：V：：C c；  c.m()；  printf("ABC = %s", abc)；  SystemPause()；  } |

命名空间外的定义和重新声明只允许在声明之后，仅在命名空间作用域内，并且只能在包含原始命名空间的命名空间（包括全局命名空间）中使用，并且必须使用qualified-id语法（C++14之后）。

|  |
| --- |
| namespace Q {  namespace V { // V原始命名空间定义  void f()； // Q：：V：：f的声明  }  void V：：f() {} // OK  void V：：g() {} // Error： g()还不是V的成员  namespace V { // V的扩展命名空间定义  void g()； // Q：：V：：g的声明  }  }  namespace R { // 不是Q的一个封闭命名空间  void Q：：V：：g() {} // Error： 不能在R内定义Q：：V：：g  }  void Q：：V：：g() {} // OK： 全局命名空间包括Q |

Names introduced by friend declarations within a non-local class X become members of the innermost enclosing namespace of X, but they do not become visible to ordinary name lookup (neither unqualified nor qualified) unless a matching declaration is provided at namespace scope, either before or after the class definition. Such name may be found through ADL which considers both namespaces and classes.

Only the innermost enclosing namespace is considered by such friend declaration when deciding whether the name would conflict with a previously declared name.

由非局部类X中的友邻声明引入的名称成为X的最内层封闭名称空间的成员，但除非在名称空间范围提供了匹配声明，否则它们对普通名称查找（既不是非限定也不是限定的） 或者在类定义之后。这个名字可以通过ADL找到，它考虑了命名空间和类。

在决定名称是否与先前声明的名称冲突时，只有最里面的封闭名称空间被这种朋友声明所考虑。

|  |
| --- |
| void h(int)；  namespace A {  class X {  friend void f(X)； // A：：f是一个友邻成员  class Y {  friend void g()； // A：：g是一个友邻成员  friend void h(int)； // A：：h是一个友邻成员，与：：h不冲突  }；  }；  // A：：f, A：：g 和 A：：h在命名空间作用域内是不可见，尽管是命名空间A的成员  X x；  void g() { // definition of A：：g  f(x)； // A：：X：：f is found through ADL  }  void f(X) {} // definition of A：：f  void h(int) {} // definition of A：：h  // A：：f, A：：g and A：：h are now visible at namespace scope  // and they are also friends of A：：X and A：：X：：Y  } |

#### 2 内联命名空间

内联命名空间就是在原始命名空间定义的基础上添加上关键字inline的命名空间。

在很多情况下，内联命名空间的成员可以被当成外层命名空间的成员那样使用。这个属性是可以传递的：如果命名空间N包含内联命名空间M，而M又包含内联命名空间O，那么O的成员就可以像使用M或N的成员那样使用。

* 命名内联命名空间的using指令隐式地插入到外层空间中（类似于不具名命名空间的隐式using指令）
* In [argument-dependent lookup](http://en.cppreference.com/w/cpp/language/adl), when a namespace is added to the set of associated namespaces, its inline namespaces are added as well, and if an inline namespace is added to the list of associated namespaces, its enclosing namespace is added as well.
* 在[依赖参数查找](#_7.6_参数依赖查找（ADL）)中，当命名空间被添加到关联的命名空间集合中时，它的内联命名空间也会被添加，并且如果将内联命名空间添加到关联命名空间列表中，它的外层命名空间也会被添加。
* 内联命名空间的每一个成员可以分别被专用，好像它们是外层命名空间的成员那样被显式地实例化或专用。
* 检查外层命名空间的限定名称查找会包含来自内联命名空间的名称，即使外层命名空间存在相同的名称。

|  |
| --- |
| { // 在C++14标准内，std：：literals和它的成员命名空间是内联的  using namespace std：：string\_literals； // 使std：：literals：：string\_literals中的操作符""s可见  auto str = "abc"s；  }  {  using namespace std：：literals； // 使std：：literals：：string\_literals：：operator""s  // std：：literals：：chrono\_literals：：operator""s都可见  auto str = "abc"s；  auto min = 60s；  }  {  using std：：operator""s； // 使std：：literals：：string\_literals：：operator""s  // std：：literals：：chrono\_literals：：operator""s都可见  auto str = "abc"s；  auto min = 60s；  } |

Note： the rule about specializations allows library versioning： different implementations of a library template may be defined in different inline namespaces, while still allowing the user to extend the parent namespace with an explicit specialization of the primary template.

注意：有关特化的规则允许库版本化：库模板的不同实现可以在不同的内联命名空间中定义，同时仍允许用户通过显式专用化主模板扩展父命名空间

#### 3 不具名命名空间

语法格式如下：

inline(optional) namespace attr(optional) { namespace-body }

attr 任意数量的属性的可选序列

这个定义被视为具有唯一名称的命名空间的定义，且在当前作用域中使用using指令指定该不具名命名空间。

|  |
| --- |
| namespace {  int i； // 定义 ：：(unique)：：i  }  void f() {  i++； // 自加 ：：(unique)：：i  }    namespace A {  namespace {  int i； // A：：(unique)：：i  int j； // A：：(unique)：：j  }  void g() { i++； } // A：：unique：：i++  }  using namespace A； // 引入所有A中名称到全局命名空间中  void h() {  i++； // error： ：：(unique)：：i and ：：A：：(unique)：：i are both in scope  A：：i++； // ok, increments ：：A：：(unique)：：i  j++； // ok, increments ：：A：：(unique)：：j  } |

Even though names in an unnamed namespace may be declared with external linkage, they are never accessible from other translation units because their namespace name is unique.（直到C++11）

Unnamed namespaces as well as all namespaces declared directly or indirectly within an unnamed namespace have internal linkage, which means that any name that is declared within an unnamed namespace has internal linkage.（C++11之后）

#### 4 using声明

Introduces a name that is defined elsewhere into the declarative region where this using-declaration appears.

|  |  |
| --- | --- |
| using typename(optional) nested-name-specifier unqualified-id ； (until C++17) | |
| using declarator-list ； (since C++17) | |
| nested-name-specifier | a sequence of names and scope resolution operators ：：, ending with a scope resolution operator. A single ：： refers to the global namespace. |
| unqualified-id | An [id-expression](http://en.cppreference.com/w/cpp/language/identifiers) |
| typename | the keyword typename may be used as necessary to resolve dependent names, when the using-declaration introduces a member type from a base class into a class template |
| declarator-list | comma-separated list of one or more declarators of the form typename(optional) nested-name-specifier unqualified-id. The last declarator may be an ellipsis, although that form is only meaningful in derived class definitions |

Using-declarations can be used to introduce namespace members into other namespaces and block scopes, or to introduce base class members into derived class definitions.

A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator. (since C++17)

对于在派生类中的使用，请参考[using declaration](http://en.cppreference.com/w/cpp/language/using_declaration)。

使用using声明这种语法把其它命名空间中的名称导入到当前命名空间中，就可以看作不同的名称了，包括在其它命名空间中的限定名称查找的名称。

|  |
| --- |
| void f()；  namespace A {  void g()；  }  namespace X {  using ：：f； // 全局f可以看作：：X：：f  using A：：g； // A：：g可以看作：：X：：g  using A：：g, A：：g； // (C++17) OK： 允许在命名空间中进行连续声明  }  void h()  {  X：：f()； // calls ：：f  X：：g()； // calls A：：g  } |

如果在使用声明从命名空间获取成员之后，命名空间被扩展并引入了相同名称的附加声明，那些附加声明就不会通过using声明变得可见（与using指令相反）。using声明命名类模板时的一个例外情况是：稍后引入的部分特化是有效可见的，因为它们的查找通过主模板进行。

|  |
| --- |
| namespace A {  void f(int)；  }  using A：：f； // ：：f 和 A：：f(int)相同    namespace A { // 命名空间扩展  void f(char)； // 没有改变：：f  }  void foo() {  f('a')； // 调用f(int),即使f(char)存在  }  void bar() {  using A：：f； // f 同时与 A：：f(int)和A：：f(char)相同  f('a')； // 调用 f(char)  } |

使用声明不能命名模板标识，名称空间或范围枚举器。 使用声明中的每个声明符都引入了一个且只有一个名称，例如枚举的using声明不会引入任何枚举器。

对相同名称的常规声明，隐藏和重载规则的所有限制均适用于使用声明：

|  |
| --- |
| namespace A {  int x；  }  namespace B {  int i；  struct g { }；  struct x { }；  void f(int)；  void f(double)；  void g(char)； // OK： function name g hides struct g  }  void func() {  int i；  using B：：i； // error： i declared twice    void f(char)；  using B：：f； // OK： f(char), f(int), f(double) are overloads  f(3.5)； // calls B：：f(double)    using B：：g；  g('a')； // calls B：：g(char)  struct g g1； // declares g1 to have type struct B：：g    using B：：x；  using A：：x； // OK： hides struct B：：x  x = 99； // assigns to A：：x  struct x x1； // declares x1 to have type struct B：：x  } |

If a function was introduced by a using-declaration, declaring a function with the same name and parameter list is ill-formed (unless the declaration is for the same function). If a function template was introduced by a using-declaration, declaring a function template with the same name, parameter type list, return type, and template parameter list is ill-formed. Two using-declarations can introduce functions with the same name and parameter list, but if a call to that function is attempted, the program is ill-formed.

|  |
| --- |
| namespace B {  void f(int)；  void f(double)；  }  namespace C {  void f(int)；  void f(double)；  void f(char)；  }  void h() {  using B：：f； // introduces B：：f(int), B：：f(double)  using C：：f； // introduces C：：f(int), C：：f(double), and C：：f(char)  f('h')； // calls C：：f(char)  f(1)； // error： B：：f(int) or C：：f(int)?  void f(int)； // error： f(int) conflicts with C：：f(int) and B：：f(int)  } |

If an entity is declared, but not defined in some inner namespace, and then declared through using-declaration in the outer namespace, and then a definition appears in the outer namespace with the same unqualified name, that definition is a member of the outer namespace and conflicts with the using-declration：（C++14后）

|  |
| --- |
| namespace X {  namespace M {  void g()； // declares, but doesn't define X：：M：：g()  }  using M：：g；  void g()； // Error： attempt to declare X：：g which conflicts with X：：M：：g()  } |

More generally, a declaration that appears in any namespace scope and introduces a name using an unqualified identifier always introduces a member into the namespace it's in and not to any other namespace. The exceptions are explicit instantiations and explicit specializations of a primary template that is defined in an inline namespace： because they do not introduce a new name, they may use unqualified-id in an enclosing namespace.

#### 5 using指令

using-directive是使用以下语法的块声明：

|  |  |
| --- | --- |
| attr(optional) using namespace nested-name-specifier(optional) namespace-name ； | |
| attr(C++11) | any number of attributes that apply to this using-directive |
| nested-name-specifier | a sequence of names and scope resolution operators ：：, ending with a scope resolution operator. A single ：： refers to the global namespace. |
| namespace-name | a name of a namespace. When looking up this name, lookup considers namespace declarations only |

using指令只允许在命名空间范围和块作用域内使用。使用using指令指定命名空间，命名空间内的每一个名称都是可见的。

using指令不会为它出现的声明区域添加任何名称（与using 声明不同），因此不会阻止声明相同的名称。

为了非限定查找的目的，using指令是具有可传递性的：如果一个作用域中包含一个用using指令指定的命名空间，这个命名空间自身又包含using指令，用来指定命名空间2，效果就好像第2个命名空间的using指令出现在第一个命名空间中一样。出现在命名空间中的顺序不会影响名称查找。

|  |
| --- |
| namespace A {  int i；  }  namespace B {  int i；  int j；  namespace C {  namespace D {  using namespace A； // A 中所有的名称被导入到全局命名空间中  int j；  int k；  int a = i； // i是B：：i, 因为A：：i被B：：i覆盖  }  using namespace D； // D 中所有的名称被导入到 C 命名空间中  // A 中所有的名称被导入到全局命名空间中  int k = 89； // OK to declare name identical to one introduced by a using  int l = k； // ambiguous： C：：k or D：：k  int m = i； // ok： B：：i hides A：：i  int n = j； // ok： D：：j hides B：：j  }  } |

如果，using指令被用来指定某些命名空间，命名空间被扩展，额外的成员或using指令被添加到该命名空间中，这些附加的成员和命名空间是可见的（与使用声明相反）。

|  |
| --- |
| namespace D  {  int d1；  void f(char)；  }  using namespace D； // 导入D：：d1, D：：f, D：：d2, D：：f, E：：e 和 E：：f到全局命名空间中    int d1； // OK： 声明时和D：：d1没有冲突  namespace E {  int e；  void f(int)；  }  // 扩展命名空间  namespace D  {  int d2；  using namespace E； // transitive using-directive  void f(int)；  }  void f() {  d1++； // 错误，：：d1 或D：：d1有歧义  ：：d1++； // OK  D：：d1++； // OK  d2++； // OK, d2 is D：：d2  e++； // OK： e is E：：e due to transitive using  f(1)； // error： ambiguous： D：：f(int) or E：：f(int)?  f('a')； // OK： the only f(char) is D：：f(char)  } |

#### 6 学习笔记

在任何命名空间内，使用using namespace std； 都会把命名空间std导入到全局命名空间中（因为全局命名空间是同时包含std和用户自定义命名空间的最近的命名空间），这会导入不想要的名称冲突。因此，并不建议使用using指令。

#### 7 例程

|  |
| --- |
| #include <vector>    namespace vec {  template< typename T >  class vector {  // ...  }；  } // of vec    int main()  {  std：：vector<int> v1； // Standard vector.  vec：：vector<int> v2； // User defined vector.    v1 = v2； // Error： v1 and v2 are different object's type.  {  using namespace std；  vector<int> v3； // Same as std：：vector  v1 = v3； // OK  }    {  using vec：：vector；  vector<int> v4； // Same as vec：：vector  v2 = v4； // OK  }    return 0；  } |

## 5.2 命名空间别名

## 5.3 左值和右值引用（已完成\*\*）

声明一个命名变量作为引用，即一个已经存在的对象或函数的别名。

### 5.3.1 语法

引用变量声明，语法如下：

|  |  |  |
| --- | --- | --- |
| & attr(可选) declarator | （1） |  |
| && attr(可选) declarator | （2） | （C++11后） |

1. 左值引用声明符
2. 右值引用声明符

declarator - 引用声明符（不包括&或&&）

attr(C++11) - 可选属性列表

引用只有初始化后，才能正确引用对象或函数：参见[引用初始化](#_6.7_引用初始化)。

没有void类型的引用，也没有引用类型的引用。

引用类型不能使用cv修饰符进行限定；声明中没有这样的语法。如果通过typedef，decltype或模板类型实参引入了cv限定符，也会被忽略。

引用不是对象，也不需要占用存储空间。但是如果有必要实现所需的语义，编译器也可以分配存储空间（例如，引用类型的非静态数据成员通常会增加所需的存储内存地址的数量来增加类的大小，也就是说，这时的引用类型的成员在类所分配的存储空间中占据一个它所引用对象的地址的大小）。

因为引用不是对象，所以没有引用数组，没有指向引用的指针，也没有指向引用的引用：

|  |
| --- |
| int& a[3]; // 错误  int&\* p; // 错误  int& &r; // 错误 |

但是在模板或typedef中的类型操作中，允许引用的引用存在，这种情况下使用“引用重叠规则”（C++11引入该规则）：右值引用的右值引用，就等于右值引用，其它组合形式等于左值引用：

|  |
| --- |
| typedef int& lref;  typedef int&& rref;  int n;  lref& r1 = n; // r1就是int&  lref&& r2 = n; // r2就是int&  rref& r3 = n; // r3就是int&  rref&& r4 = 1; // r4就是int&& |

（当在函数模板中使用T&&引用时，“引用重叠规则”和“模板实参推导”规则产生了std::forward）

### 5.3.2 左值引用

其实，左值引用就是已经存在的对象的别名（可选地具有不同的cv资格）：

|  |
| --- |
| #include <iostream>  #include <string>    int main()  {  std::string s = "Ex";  std::string& r1 = s;  const std::string& r2 = s;    r1 += "ample"; // 修改s  // r2 += "!"; // 错误：无法通过引用修改const常量  std::cout << r2 << '\n'; // 打印s，输出"Example"  } |

它们也可以用于在函数调用中实现传递引用语义：

|  |
| --- |
| #include <iostream>  #include <string>    void double\_string(std::string& s)  {  s += s; // “s” 和main()中的“str”是相同的对象  }    int main()  {  std::string str = "Test";  double\_string(str);  std::cout << str << '\n';  } |

当函数的返回类型是左值引用时，函数调用表达式变成左值表达式：

|  |
| --- |
| #include <iostream>  #include <string>    char& char\_number(std::string& s, std::size\_t n)  {  return s.at(n); // string::at() 返回对某个字符的引用  }    int main()  {  std::string str = "Test";  char\_number(str, 1) = 'a'; // 函数调用成为一个左值表达式，所以能够被赋值  std::cout << str << '\n';  } |

### 5.3.3 右值引用

右值引用可用于扩展临时对象的生命周期（注意，对const常量的左值引用也可以延长临时对象的生命周期，但不能通过它们修改）：

|  |
| --- |
| #include <iostream>  #include <string>    int main()  {  std::string s1 = "Test";  // std::string&& r1 = s1; // 错误：右值引用不能绑定到左值上    const std::string& r2 = s1 + s1; // 可以：对const常量的左值引用，扩展其生命周期  std::string& r2 = s1 + s1; // 错误：非const引用初始化不正确  // r2 += "Test"; // 错误：不能通过const常量的左值引用，修改其内容    std::string&& r3 = s1 + s1; // 可以：右值引用扩展其生命周期  r3 += "Test"; // 可以: 可以通过右值引用修改其引用的对象。  std::cout << r3 << '\n';  } |

更重要的是，当函数同时具有右值引用和右值引用重载时，右值引用重载绑定到右值（包括纯右值和消亡值），而左值引用重载绑定左值：

|  |
| --- |
| #include <iostream>  #include <utility>    void f(int& x)  {  std::cout << "lvalue reference overload f(" << x << ")\n";  }  void f(const int& x)  {  std::cout << "lvalue reference to const overload f(" << x << ")\n";  }  void f(int&& x)  {  std::cout << "rvalue reference overload f(" << x << ")\n";  }  int main()  {  int i = 1;  const int ci = 2;  f(i); // calls f(int&)  f(ci); // calls f(const int&)  f(3); // calls f(int&&)  // would call f(const int&) if f(int&&) overload wasn't provided  f(std::move(i)); // calls f(int&&)    // rvalue reference variables are lvalues when used in expressions  int&& x = 1;  f(x); // calls f(int& x)  f(std::move(x)); // calls f(int&& x)  } |

这允许移动构造函数，移动赋值运算符和其它移动相关函数（例如vector::push\_back()在适当时自动选择。

因为右值引用可以绑定到xvalue，所以它们可以引用非临时对象

|  |
| --- |
| int i2 = 42;  int&& rri = std::move(i2); // 直接绑定到i2 |

这使得可以移出作用域内不再需要的对象

|  |
| --- |
| std::vector<int> v{1,2,3,4,5};  std::vector<int> v2(std::move(v)); // 把一个右值引用绑定到v上  assert(v.empty()); |

在这儿，std::move()就是把左值或者右值转换成右值引用。

查看下面的代码，理解std::move()的语义。

|  |
| --- |
| #include <iostream>  #include <utility>  #include <vector>  #include <string>  int main()  {  std::string str = "Hello";  std::vector<std::string> v; //调用常规的拷贝构造函数，新建字符数组，拷贝数据  v.push\_back(str);  std::cout << "After copy, str is \"" << str << "\"\n";  //调用移动构造函数，掏空str，掏空后，最好不要使用str  v.push\_back(std::move(str));  std::cout << "After move, str is \"" << str << "\"\n";  std::cout << "The contents of the vector are \"" << v[0]  << "\", \"" << v[1] << "\"\n";  } |

运行结果为

|  |
| --- |
| After copy, str is "Hello"  After move, str is ""  The contents of the vector are "Hello", "Hello" |

理解：

由于STL里面默认的库已经支持右值引用，也有所谓的移动构造函数如下面的形式A (A&& a){} 这里不能使用const A&& a，因为需要改变a。移动构造函数主要的用途是，当你不需要在使用一个变量的时候，可以直接通过该构造函数来实现把该变量的数据转换到另一个变量中，省去调用默认的赋值构造或者拷贝构造函数带来额外的开销，如string类在赋值或者拷贝构造函数中会声明char数组来存放数据，然后把原string中的 char 数组被析构函数释放，如果a是一个临时变量，则上面的拷贝，析构就是多余的，完全可以把临时变量a中的数据直接“转移”到新的变量下面即可。

另外，需要注意的是str的内容此时已经被移除了，所以，最好不要再调用它了。

### 5.3.4 转发引用（C++11）

如果把一个右值引用作为函数的实参传递给传输的形参，根据C++标准的语义，这个参数就变成了一个左值。所以，在这个函数里再使用这个参数的时候，比如说，再用该参数调用另一个函数，就是左值引用了。可能往往与我们期待的右值引用相违背。于是，在C++11中，加入了转发引用的方法。

转发引用是一种特殊类型的引用，通过函数std::forward，它保留了函数实参的值类型。

1) 查看下面的代码进行理解

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  int main()  {  std::string A("abc"); // A是一个左值  std::string&& Rval = std::move(A); // Rval是右值引用  std::string B(Rval); // 这是拷贝，不是移动  std::cout << A << std::endl; // 输出"abc"  std::string C(std::forward<std::string>(Rval)); // 移动.  std::cout << A << std::endl; // 输出""  return 0;  } |

代码分析：

上面的代码中，首先，通过拷贝赋值构造函数实例化一个字符串对象A。然后通过移动语义赋值给右值引用。然后就是通过拷贝赋值得到对象B，此时，打印A，输出"abc"。然后通过转发引用将右值引用Rval指向的值，在std::string的移动构造函数调用作用下，赋值给新的对象C；而右值引用指向的值的生命周期也结束了。所以，此时再打印A，只能得到空字符串。

2) auto&& 引用在大括号括起来的初始化列表时，是个例外。

|  |
| --- |
| auto&& vec = foo(); // foo() 可以是左值或右值，vec是转发引用  auto i = std::begin(vec); // 没问题  (\*i)++; // 没问题  g(std::forward<decltype(vec)>(vec)); // 转发，保留值的类型    for(auto&& x:f()) {  // x 是一个转发引用：这是使用循环的安全方式  }    auto&& z = {1,2,3}; // 不是转发引用（初始化列表的特例） |

而std::forward的理解，我们可以参考下面的代码：

|  |
| --- |
| template<typename T>  void PrintT(T& t)  {  std::cout << "lvaue reference" << std::endl;  }  template<typename T>  void PrintT(T&& t)  {  std::cout << "rvalue reference" << std::endl;  }  template<typename T>  void TestForward(T&& v)  {  PrintT(v);  PrintT(std::forward<T>(v));  PrintT(std::move(v));  }  int main(void)  {  int x = 1;  // 1是右值，所以结果应该是左值引用，右值引用，右值引用  TestForward(1);  // x是左值，所以结果应该是左值引用，左值引用，右值引用  TestForward(x);  // std::forward<int>(x)作为实参时是右值，所以结果应该是左值引用，右值引用，右值引用  TestForward(std::forward<int>(x));  } |

### 5.3.5 悬空引用

尽管引用初始化后始终引用有效的对象或函数，但有时候会有这样的程序，引用对象的[生命周期](#_1.5.3_生命周期)结束了，但引用仍可访问（悬空状态）。访问这样的引用是未定义的行为。一个常见的例子是一个函数返回一个自动变量的引用：

|  |
| --- |
| std::string& f()  {  std::string s = "Example";  return s; // 退出s的作用范围: // 调用析构函数，它的内存空间被释放  }  std::string& r = f(); // r将会是悬空引用  std::cout << r; // 未定义的行为：读取一个悬空引用  std::string s = f(); // 未定义的行为：用悬空引用来拷贝初始化 |

请注意，对const常量的右值引用和左值引用可延长临时对象的生命周期（有关规则和异常，请参阅[引用初始化](#_6.7_引用初始化)的[临时对象的生命周期](#_6.7.3_临时对象的生命周期)）。

如果引用的对象被销毁（例如，通过显式地调用析构函数），但是存储空间没有被释放，那么对于过期对象的引用可能以有限的方式使用，并且如果该对象被重新创建相同的存储（有关详细信息，请参阅[生命周期外的访问](#_1.5.3.3在生命周期之外访问)）。

## 5.4 指针和数组

### 5.4.1 指针

#### 5.4.1.1 语法

#### 5.4.1.2 指针

#### 5.4.1.3 对象指针

#### 5.4.1.4 void指针

#### 5.4.1.5 函数指针

#### 5.4.1.6 成员指针

1. 数据成员指针

2. 函数成员指针

#### 5.4.1.7 NULL指针

#### 5.4.1.8 const指针

### 5.4.2 数组

#### 5.4.2.1 语法

#### 5.4.2.2 赋值

#### 5.4.2.3 数组向指针的退化

#### 5.4.2.4 多维数组

#### 5.4.2.5 无界数组

#### 5.4.2.6 数组右值

## 5.5 结构化绑定（C++17后）

将指定的名称绑定到初始值设定项的子对象或元素。

像引用一样，结构化绑定是现有对象的别名。与引用不同的是，结构化绑定的类型不一定是引用类型。

### 5.5.1 语法解释

### 5.5.2 举例说明

案例1：绑定数组

标识符列表中的每个标识符都成为引用数组的相应元素的左值的名称。标识符的数量必须等于数组元素的数量。

每个标识符的引用类型是数组元素类型。请注意，如果数组类型E是cv限定的，则其元素类型也是如此。

|  |
| --- |
| int a[2] = {1,2};  auto [x,y] = a; // 创建数组e[2]，赋值a到e，然后x引用e[0]，y引用e[1]  auto& [xr, yr] = a; // xr引用a[0]，yr引用a[1] |

案例2：绑定元组类型

## 5.6 枚举

### 5.6.1 枚举声明

### 5.6.2 没有作用域的枚举

|  |  |  |
| --- | --- | --- |
| enum name {枚举项1=常量表达式,枚举项2=常量表达式,...} | （1） |  |
| enum name : 类型{枚举项1=常量表达式,枚举项2=常量表达式,...} | （2） | C++11后 |
| enum name : 类型; | （3） | C++11后 |

* 声明一个不具有作用域的枚举型，它的隐含类型是不固定的（在这种情况下，隐含类型是执行时定义的整型，这个整型能够表示所有的枚举值；这个类型不大于int型，除非枚举器的值不适合int或者unsigned int。如果枚举器列表是空的，隐含类型就好像是这个枚举器具有单个枚举值0一样。）
* 声明一个不具有作用域的枚举型，它的隐含类型是固定的。
* 3）不具有作用域的枚举的不透明枚举声明必须指定隐含类型。

每个枚举器都变成枚举类型的名称常量（即名称），在封闭范围内可见，并且可以在需要常量时使用。

|  |
| --- |
| enum Color { red, green, blue };  Color r = red;  switch(r)  {  case red : std::cout << "red\n"; break;  case green: std::cout << "green\n"; break;  case blue : std::cout << "blue\n"; break;  } |

每个枚举值都与隐含类型的值相关联。当在枚举器列表中提供初始化器时，枚举器的值由那些初始化器定义。如果第一个枚举项没有初始化，则相关的值为零。对于那些没有初始化的枚举项，关联值是前一个枚举项的值加1。

### 5.6.3 具有作用域的枚举（C++11之后）

|  |  |
| --- | --- |
| enum struct|class 枚举名称{枚举项1=常量表达式,枚举项2=常量表达式,...} | （1） |
| enum struct|class枚举名称:类型{枚举项1=常量表达式,枚举项2=常量表达式,...} | （2） |
| enum struct|class枚举名称; | （3） |
| enum struct|class枚举名称:类型; | （4） |

* 声明具有作用域的枚举型，它的隐含类型是int（关键字class和struct完全等价）；
* 声明具有作用域的枚举型，它的隐含类型是type；
* 具有作用域的枚举的不透明声明，其隐含类型是int；
* 具有作用域的枚举的不透明声明，其隐含类型是type；

每个枚举项都成为枚举类型的名称常量（即名称），其被包含在枚举作用域内，可以使用作用域解析符::进行访问。虽然可以使用static\_cast来获取枚举数的数值，但是没有从作用域枚举数到整型的隐式转换。

|  |
| --- |
| enum class Color { red, green = 20, blue };  Color r = Color::blue;  switch(r)  {  case Color::red : std::cout << "red\n"; break;  case Color::green: std::cout << "green\n"; break;  case Color::blue : std::cout << "blue\n"; break;  }  // int n = r; // 错误：没有作用域枚举值向int型的隐式转换  int n = static\_cast<int>(r); // 可以，n=21 |

如果下面的条件被满足，没有作用域和有作用域的枚举都能使用整型进行初始化，而不用进行转换。（以下内容适用于C++17之后）

* 初始化是直接列表初始化；
* 初始化列表只有一个元素；
* 枚举的隐含类型是固定的（包括没有作用域的枚举和有作用域的枚举）；
* the conversion is non-narrowing

This makes it possible to introduce new integer types (e.g. SafeInt) that enjoy the same existing calling conventions as their underlying integer types, even on ABIs that penalize passing/returning structures by value.

|  |
| --- |
| enum byte : unsigned char {}; // byte 是一个新的整形  byte b { 42 }; // 在C++17中是OK的 (直接列表初始化)  byte c = { 42 }; // 错误  byte d = byte{ 42 }; // 在C++17中是OK的；和b的值相等  byte e { -1 }; // 错误    struct A { byte b; };  A a1 = { { 42 } }; // 错误  A a2 = { byte{ 42 } }; // 在C++17中是OK的    void f(byte);  f({ 42 }); // 错误    enum class Handle : std::uint32\_t { Invalid = 0 };  Handle h { 42 }; // 在C++17中是OK的 |

注意：可以在类中声明枚举，借助类的作用域；

### 5.6.4 举例说明

|  |
| --- |
| #include <iostream>    // 枚举定义为16位  enum smallenum: int16\_t  {  a,  b,  c  };    // color可以是red(0)，yellow(1)，green(20)，或blue(21)  enum color  {  red,  yellow,  green = 20,  blue  };    // altitude是altitude::high或altitude::low  enum class altitude: char  {  high='h',  low='l', // C++11允许额外的逗号  };    // 常量d是0，常量e是1，常量f是3  enum  {  d,  e,  f = e + 2  };    //枚举类型(both scoped and unscoped)都可以有重载运算符  std::ostream& operator<<(std::ostream& os, color c)  {  switch(c)  {  case red : os << "red"; break;  case yellow: os << "yellow"; break;  case green : os << "green"; break;  case blue : os << "blue"; break;  default : os.setstate(std::ios\_base::failbit);  }  return os;  }    std::ostream& operator<<(std::ostream& os, altitude al)  {  return os << static\_cast<char>(al);  }    int main()  {  color col = red;  altitude a;  a = altitude::low;    std::cout << "col = " << col << '\n'  << "a = " << a << '\n'  << "f = " << f << '\n';  } |

## 5.7 存储周期和链接

### 5.7.1 存储类关键字

存储类关键字是名称声明语法的decl-specifier-seq的一部分。和名称的作用域一起，控制着名称的两个独立属性，自动存储期和链接属性。

* auto 自动存储期。（C++11之前适用）
* register 自动存储期。另外，提醒编译器把对象放入处理器的寄存器中。（C++17之前适用，现已被废弃）
* static 静态或线程存储期，内部链接属性。
* extern 静态或线程存储期，外部链接属性。
* thread\_local 线程存储期。（C++11之后适用）

一次只能一个存储类关键字出现在声明语句中，thread\_local是个例外，需要与static和或者extern结合使用。（C++11之后适用）

### 5.7.2 解释

* 关键字auto只能声明在块作用域或函数参数列表中的对象。它代表着其默认是自动存储期。在C++11中，这个关键字的意义被改变。
* 关键字register也只能声明在块作用域或函数参数列表中的对象。 默认是自动存储期。另外，这个关键字提示代码优化器保存该变量的值在CPU寄存器里。C++11放弃了这个关键字。
* 关键字static，能够在对象的声明（除函数列表外），函数的声明（除在块作用域）和不具名联合体声明里使用。当用在类成员上时，它声明了一个静态成员。当用在对象声明上时，它指定了静态存储期（如果和thread\_local联合使用除外）。当用在命名空间作用域内时，它指定了内部链接属性。
* 关键字extern只被允许用在变量和函数的声明上（除了类成员或函数参数）。它指定了外部链接属性，且不会影响存储期，但是它不能被用在一个具有自动存储期的对象身上，所以，所有的extern对象具有static或thread存储周期。另外，使用了extern且没有初始化的变量声明不是一个定义。
* 关键字thread\_local被允许声明命名空间范围和块作用域内的对象，及静态数据成员。它指明，对象具有线程存储周期。可以和static或extern关键字一起使用，指明内部或者外部链接属性（除了static数据成员，其余的总是具有外部链接属性），但是添加的static关键字不会影响其存储周期。

### 5.7.3 存储期

所有的对象都具有下面这些存储类型中的一种：

* automatic

对象在代码块开始时被分配，离开时收回分配的存储空间。所有的局部对象都有这种存储周期，除非，它们被声明为static，extern或thread\_local。

* static

当程序开始运行时分配对象的存储空间，程序结束时收回对象的存储空间。只允许一个对象实例存在。所有声明在命名空间的对象（包括全局命名空间），前面加上static，或者extern关键字的都有这种存储周期。

* thread

线程开始分配对象，线程结束收回分配给对象的存储空间。每个线程拥有这个对象唯一的实例。只有使用关键字thread\_local声明的对象才有这种存储周期。关键字thread\_local 可以和static或extern结合使用，以调整链接属性。

* dynamic

当使用动态内存分配函数请求分配或者回收对象的存储空间时才会使用。

### 5.7.4 链接属性

名称泛指对象，引用，函数，类型，模板，命名空间，或数值（枚举器），都可以有链接属性。如果一个名称具有链接属性，那么在另一个作用域内声明而引入的相同名称就会被引用为同一个实体。如果在几个作用域内声明了具有相同名称的变量，函数，或另一个实体，但是又没有有效的链接属性，那么就会产生几个实体实例。

链接类型可以被分为下面三种：

（1）无链接。

这种方式适用于名称在它的作用域内的情况。下面的几种情况具有非链接属性：

* 名称没有显式地使用extern关键字声明（无关static修饰符）；
* 局部类和它的成员函数；
* 块作用域内声明的其它名称，例如typedef，enum等声明的名称；

（2）内部链接。

在当前的编译单元里能够被所有的作用域引用的名称。命名空间作用范围内声明的下面中的任何一种名称都具有内部链接属性：

* static声明的变量，函数，和函数模板；
* 没有使用extern声明或者之前也没有被声明为具有外部链接属性的非易失性（non-volatile）非内联常量限定的变量（包括constexpr）；
* 不具名联合体的数据成员；

另外，在不具名命名空间或者不具名命名空间内部的命名空间里声明的所有变量，即使明确使用extern声明，也是内部链接属性。

（3）外部链接

可被其它编译单元参考引用的名称，就具有外部链接属性。具有外部链接属性的变量和函数也具有语言链接属性，这使得链接不同程序语言写的编译单元成为可能。

任何在命名空间里声明的下列变量都具有外部链接属性，除非，命名空间是不具名的或者被一个不具名命名空间包含（C++11之后）。

* 上面没有列出的变量和函数（也就是说，没有被声明为static函数，命名空间范围内的非const变量没有被声明为static，任何声明为extern的变量）
* 枚举和枚举器
* 类名，它们的成员函数，静态数据成员（const与否），嵌套类和枚举类型变量，类体内首次使用友邻声明的函数
* 上面没有列出的所有模板变量（就是说，声明为static的非函数模板）

首次声明在块作用域内的下列变量中任何一种都有外部链接属性：

* 声明为extern的变量
* 函数变量

### 5.7.5 静态局部变量

使用限定符static声明在块作用域内的变量具有static存储期，只有当第一次执行经过它们的声明时被初始化（除非它们的初始化是0或常量初始化，这种初始化可以在进入块作用域之前就已经完成）。在所有后面的调用中，声明都会被跳过，不执行。

如果初始化过程出现异常，那么不认为变量被初始化，再一次尝试控制经过声明语句时，还会初始化。

如果多个线程同时尝试初始化相同的静态变量，初始化也只会进行一次（对于使用std：：cal\_once的任意函数都能获得相似的行为）。

注意：这个功能的通常实现就是使用双重检查锁定模式，它可以减少已经初始化为局部静态和单个非原子boolean的比较产生的系统开销。

当程序exit时，调用块作用域的析构函数，但前提是初始化成功。

对于同一个内联函数（也许是隐含内联）的所有定义里的局部静态对象，都会被一个编译单元定义的相同的对象引用。

### 5.7.6 注意

在C语言中，在顶层命名空间作用域（相当于C的文件范围）内的，是const且没有extern修饰的名称具有外部属性，但是在C++中却是内部链接属性。

在C里，寄存器变量的地址不能获取，但是在C++中，变量声明为register和没有任何存储类关键字修饰是没有什么区别的。（C++11之前）

C++中，不像C，变量不能声明为register。（C++17之后）

具有内部或外部链接属性的thread\_local型变量的名称可以被不同的实例引用，依赖于代码是否在同一个或者不同的线程中执行。

关键字extern也可以指定语言链接属性和明确的模板实例声明，但是它不是存储类限定符（除非，声明被直接包含在语言链接指定中，在这种情况时，声明被像包含extern限定符一样对待）。

在C++语法中，关键字mutable是存储类限定符，尽管它不会影响存储周期或链接属性。

现在这段是不完整的，因为在同一个编译单元重新声明的规则。

存储类限定符，对于thread\_local是个例外，不允许明确的指定和明确的实例。

template <class T> struct S {

thread\_local static int tlm;

};

template <> thread\_local int S<float>::tlm = 0; // "static" 没有出现在这里

### 5.7.7 关键字

auto, register, static, extern, thread\_local

### 5.7.8 举例说明

#include <iostream>

#include <string>

#include <thread>

#include <mutex>

thread\_local unsigned int rage = 1;

std::mutex cout\_mutex;

void increase\_rage(const std::string& thread\_name)

{

++rage; // 锁外修改是没问题的;这是一个thread-local 变量

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for " << thread\_name << ": " << rage << '\n';

}

int main()

{

std::thread a(increase\_rage, "a"), b(increase\_rage, "b");

{

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for main: " << rage << '\n';

}

a.join();

b.join();

}

可能的输出：

Rage counter for a: 2

Rage counter for main: 1

Rage counter for b: 2

## 5.8 语言链接

为由两个不同的语言写的模块提供链接。

### 5.8.1 语法

|  |  |
| --- | --- |
| extern 常量字符串 { 声明序列(可选) } | （1） |
| extern常量字符串 声明 | （2） |

* 将符合语言规范的字符串文字应用于所有具有外部链接属性的函数类型，函数名称和声明序列中声明的具有外部链接的变量。
* 将符合语言规范的字符串文字应用于单个声明或定义。
* 常量字符串 - 所需链接语言的名称
* 声明序列 - 一些列的声明，可能包含嵌套的链接规范
* 声明 - 声明

### 5.8.2 解释

Every function type, every function name with external linkage, and every variable name with external linkage, has a property called language linkage. Language linkage encapsulates the set of requirements necessary to link with a module written in another programming language: calling convention, name mangling algorithm, etc.

Only two language linkages are guaranteed to be supported:

1) "C++", the default language linkage.

2) "C", which makes it possible to link with functions written in the C programming language, and to define, in a C++ program, functions that can be called from the modules written in C.

|  |
| --- |
| extern "C" {  int open(const char \*pathname, int flags); // C 函数声明  }  int main()  {  int fd = open("test.txt", 0); // C++程序中调用C函数  }  // C程序也可以调用C++代码  extern "C" void handler(int) {  std::cout << "Callback invoked\n";  } |

Since language linkage is part of every function type, pointers to functions maintain language linkage as well. Language linkage of function types (which represents calling convention) and language linkage of function names (which represents name mangling) are independent of each other:

|  |
| --- |
| extern "C" void f1(void(\*pf)()); // 声明函数f1为C函数  // 返回void 且调用一个指向C函数的指针  // 指针指向的函数返回void且没有参数  extern "C" typedef void FUNC(); // 声明FUNC 作为C函数类型，且返回void，没有参数  FUNC f2; // 名称f2具有C++链接属性，但是他的类型却是C函数  extern "C" FUNC f3; // 名称f3 具有C链接属性且他的类型具有C 函数类型void()  void (\*pf2)(FUNC\*); // pf2具有C++ 链接属性且它的类型是指向C++函数的指针  // 它的返回类型是void，且有一个参数是指向C函数的指针。  extern "C" {  static void f4(); // 函数名称f4具有内部链接属性(无关语言)  // 但是函数的类型是C链接属性  } |

Two functions with the same name and the same parameter list in the same namespace cannot have two different language linkages (note, however, that linkage of a parameter may permit such overloading, as in the case of std::qsort and std::bsearch). Likewise, two variables in the same namespace cannot have two different language linkages.

Special rules for "C" linkage

1) When class member declarations and member function type declarations appear in a "C" language block, their linkage remains "C++".

2) When two functions with the same unqualified name are declared in different namespaces, and both have "C" language linkage, the declarations refer to the same function.

3) When two variables with "C" language linkage and the same name that appear in different namespaces, they refer to the same variable.

4) A "C" variable and a "C" function cannot have the same name, regardless if they are defined in the same or different namespaces.

Notes

Language specifications can only appear in namespace scope.

The braces of the language specification do not establish a scope.

When language specifications nest, the innermost specification is the one that is in effect.

A function can be re-declared without a linkage specification after it was declared with a language specification, the second declaration will reuse the first language linkage. The opposite is not true: if the first declaration has no language linkage, it is assumed "C++", and redeclaring with another language is an error.

A declaration directly contained in a language linkage specification is treated as if it contains the extern specifier for the purpose of determining the linkage of the declared name and whether it is a definition.

|  |
| --- |
| extern "C" int x; // a declaration and not a definition  // The above line is equivalent to extern "C" { extern int x; }    extern "C" { int x; } // a declaration and definition |

extern "C" makes it possible to include header files containing declarations of C library functions in a C++ program, but if the same header file is shared with a C program, extern "C" (which is not allowed in C) must be hidden with an appropriate #ifdef, typically \_\_cplusplus:

|  |
| --- |
| #ifdef \_\_cplusplus  extern "C" int foo(int, int); // C++ compiler sees this  #else  int foo(int, int); // C compiler sees this  #endif |

## 5.9 内嵌汇编

asm声明允许在C++程序中嵌入汇编语言源代码。这个声明是有条件支持和执行定义的，这意味着它可能不存在，即使执行时提供，它也没有固定的含义。

1 语法

|  |
| --- |
| asm(*字符串文本*); |

2 解释

*字符串文本*通常是一个用汇编语言编写的短程序，只要执行该声明就会执行该程序。 不同的C++编译器对asm声明有不同的规则，对于与周围C++代码的交互有不同的约定。

与其它块声明一样，此声明可以出现在块（函数体或其它复合语句）中，并且与所有其它声明一样，此声明也可以出现在块之外。

本节不完整

原因：在GCC扩展汇编语法中写一个注释，因为现在Intel，IBM，Sun（从第12版开始）支持，等等

3 举例

演示GCC编译器提供的两种内联汇编语法。该程序只能在Linux下的x86\_64平台上正常工作。

|  |
| --- |
| #include <iostream>    extern "C" int func();  // 函数func的定义使用汇编语言编写  // 原始字符串文字可能非常有用  asm(R"(  .globl func  .type func, @function  func:  .cfi\_startproc  movl $7, %eax  ret  .cfi\_endproc  )");  int main()  {  int n = func();  // 扩展内联汇编  asm ("leal (%0,%0,4),%0"  : "=r" (n)  : "0" (n));  std::cout << "7\*5 = " << n << std::endl; // 冲刷  // 标准内联汇编  asm ("movq $60, %rax\n\t" // Linux上的退出系统调用数字  "movq $2, %rdi\n\t" // 这个程序返回 2  "syscall");  } |

输出：

|  |
| --- |
| 7\*5 = 35 |

## 5.10 CV类型限定符–常量表达式

### 5.10.1 const和volatile类型限定符

const和 volatile出现在任何类型说明符中，包括[声明语法](#_5_声明)的decl-specifier-seq中，用来指定正在被声明的对象或者正在命名的类型的常量性（constness）或者易变性（volatile）。

（1）const 定义类型为常量；

（2）volatile 定义类型为易失型；

（3）mutable 适用于非引用非const类型的非静态类成员，并指定该成员不影响类的外部可见状态（如常用于互斥，内存高速缓存，延迟评估（lazy evaluation）和访问工具（access instrumentation））。Const类实例的mutable成员是可以修改的。（注意：C++语法虽然把mutable作为存储类说明符，但是它并不影响存储类）。

### 5.10.2 解释

对于任何类型T（包括[不完整类型](#_1.4.6_不完整类型)），除[函数类型](#_7_函数)或引用类型之外，C++类型系统中还有三种不同类型：const限定的T，volatile限定的T和const和volatile限定的T.

注意：数组类型被认为与它们的元素类型具有相同的cv限定。

首次创建对象时，使用的cv限定符（可能是decl-specifier-seq的一部分，或者是声明中声明符的一部分，或者是[new表达式](#_4.4.7_new_-)中的type-id的一部分）会确定对象的常量或易变性，如下所示：

* **const 对象** – const类型的对象，或一个const对象的非mutable子对象。这样的对象不能被修改：试图直接这样做会发生编译时错误，并且试图间接地这样做（例如，通过引用或指向非const类型的指针修改const对象）会导致未定义的行为；
* **volatile 对象** - an object whose type is *volatile-qualified*, or a subobject of a volatile object, or a mutable subobject of a const-volatile object. Every access (read or write operation, member function call, etc.) made through a glvalue expression of volatile-qualified type is treated as a visible side-effect for the [purposes of optimization](http://en.cppreference.com/w/cpp/language/as_if) (that is, within a single thread of execution, volatile accesses cannot be optimized out or reordered with another visible side effect that is [sequenced-before](http://en.cppreference.com/w/cpp/language/eval_order) or sequenced-after the volatile access. This makes volatile objects suitable for communication with a [signal handler](http://en.cppreference.com/w/cpp/utility/program/signal), but not with another thread of execution, see [std：：memory\_order](http://en.cppreference.com/w/cpp/atomic/memory_order)). Any attempt to refer to a volatile object through a non-volatile [glvalue](http://en.cppreference.com/w/cpp/language/value_category#glvalue) (e.g. through a reference or pointer to non-volatile type) results in undefined behavior.
* **volatile对象** - 类型为volatile的对象，volatile对象的子对象或const-volatile对象的mutable子对象。 通过volatile类型的[泛左值](#_4.1.2.1_泛左值（glvalue）)表达式进行的每次访问（读取或写入操作，成员函数调用等）都被视为可见副作用，用于优化目的（即在单个执行线程内，volatile 在易失性访问之后，访问不能被优化或重排序的另一个可见的副作用 - 在被排序之前或排序 - 这使得易失性对象适合与信号处理程序进行通信，但不与另一个执行线程通信，请参阅std :: memory\_order）。 任何通过非易失性glvalue引用易失性对象的尝试（例如通过引用或指向非易失性类型的指针）都会导致未定义的行为。
* **const volatile object** - an object whose type is *const-volatile-qualified*, a non-mutable subobject of a const volatile object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object. Behaves as both a const object and as a volatile object.

对于任何类型T（包括不完整的类型），除了函数类型或引用类型，C++类型系统还有三种不同的类型：const限定类型，volatile限定类型，常量volatile限定类型

### 5.10.3 常量表达式限定符（C++11之后）

constexpr - 指明变量或函数的值可以出现在[常量表达式](#_4.3_常量表达式)中

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given). A constexpr specifier used in an object declaration implies const. A constexpr specifier used in a function or static member variable (since C++17) declaration implies inline.

constexpr限定符声明可以在编译时计算的函数或变量值。这样的变量和函数可以用在只允许编译时间常量表达式的地方（只要给出适当的函数参数）。 对象声明中使用的constexpr说明符暗示const。 在函数或静态成员变量（自C ++ 17）声明中使用的constexpr说明符暗含内联。

A constexpr variable must satisfy the following requirements:

一个constexpr变量必须满足以下要求：

* 它的类型必须是文本类型（LiteralType）；
* 它必须被立即初始化；
* the [full-expression](#_4.2_操作数计算顺序) of its initialization, including all implicit conversions, constructors calls, etc, must be a 常量表达式；

一个constexpr函数必须满足以下要求：

不能是虚函数

返回类型必须是文本类型（LiteralType）

它的每个参数必须是文本类型（LiteralType）

there exists at least one set of argument values such that an invocation of the function could be an evaluated subexpression of a core constant expression (for constructors, use in a constant initializer is sufficient) (since C++14). No diagnostic is required for a violation of this bullet.

至少存在一组参数值，使得函数的调用可以是核心常量表达式的评估子表达式（对于构造函数，在常量初始化器中使用就足够了）（自C ++ 14以来）。 违反此项目不需要诊断。

## 5.11 decltype-auto(C++11)

### 5.11.1 decltype操作符

检查实体的声明类型或表达式的类型和值类别。

在C++中，作为操作符，用于查询表达式的数据类型。decltype 在C++11标准制定时引入，主要是为泛型编程而设计，以解决泛型编程中，由于有些类型由模板参数决定，而难以（甚至不可能）表示之的问题。

从语义上说，decltype的设计适合于通用库编写者与编程新手。总体上说，对于目标对象或函数，由decltype推导出的类型与源码中的定义可精确匹配。而正如sizeof操作符一样，decltype亦不需对操作数求值。

#### 5.11.1.1 语法

|  |  |  |
| --- | --- | --- |
| decltype (实体) | （1） | C++11之后 |
| decltype (表达式) | （2） | C++11之后 |

#### 5.11.1.2 解释

* 如果参数是一个未命名的id-表达式命名的[结构化绑定](#_5.5_结构化绑定（C++17后）)，则decltype会生成引用的类型（在结构化绑定声明的规范中描述）。
* 如果参数是一个非私有的id表达式或一个未私有的类成员访问表达式，则decltype产生由该表达式命名的实体的类型。 如果没有这样的实体，或者如果参数指定了一组重载函数，则该程序是格式不正确的。
* 如果参数是T类型的任何其它表达式，
  + 如果表达式的值类别是xvalue，则decltype产生T &&；
  + 如果表达式的值类别是左值，则decltype产生T＆；
  + 如果表达式的值类别是prvalue，则decltype产生T。

If expression is a function call which returns a prvalue of class type or is a comma expression whose right operand is such a function call, a temporary object is not introduced for (until C++17)materialized from (since C++17) that prvalue. The class type need not be complete or have an available destructor. This rule doesn't apply to sub-expressions: in decltype(f(g())), g() must have a complete type, but f() need not.

Note that if the name of an object is parenthesized, it is treated as an ordinary lvalue expression, thus decltype(x) and decltype((x)) are often different types.

decltype is useful when declaring types that are difficult or impossible to declare using standard notation, like lambda-related types or types that depend on template parameters.

如果expression是一个函数调用，它返回一个类类型的prvalue，或者是一个逗号表达式，它的右操作数就是这样一个函数调用，那么临时对象不会被引入（从C ++ 17开始）（直到C ++ 17） 那个价值。 类类型不需要是完整的或具有可用的析构函数。 此规则不适用于子表达式：在decltype（f（g（）））中，g（）必须具有完整类型，但f（）不需要。

请注意，如果一个对象的名称被加上括号，它将被视为普通的左值表达式，因此decltype（x）和decltype（（x））通常是不同的类型。

当使用标准符号声明难以或不可能声明的类型时，decltype非常有用，例如与lambda相关的类型或依赖于模板参数的类型。

#### 5.11.1.3 举例

|  |
| --- |
| #include <iostream>    struct A { double x; };  const A\* a;    decltype(a->x) y; // y的类型是double  decltype((a->x)) z = y; // z的类型是const double& (左值表达式)    template<typename T, typename U>  auto add(T t, U u) -> decltype(t + u) // 返回类型依赖模板参数//返回类型可以被推导（C++14之后）  {  return t+u;  }    int main()  {  int i = 33;  decltype(i) j = i \* 2;    std::cout << "i = " << i << ", " << "j = " << j << '\n';    auto f = [](int a, int b) -> int  {  return a \* b;  };    decltype(f) g = f; // lambda函数的类型是唯一且未命名的  i = f(2, 2);  j = g(3, 3);    std::cout << "i = " << i << ", " << "j = " << j << '\n';  } |

输出结果：

|  |
| --- |
| i = 33, j = 66  i = 4, j = 9 |

## 5.12 alignas(C++11)

## 5.13类型别名

### 5.13.1 typedef修饰符

创建一个别名，可以在任何地方用来代替（可能是复杂的）类型名称。而不只是简单的宏替换。

1. 解释

说明符typedef在声明的decl-specifier-seq中使用时指定声明是typedef声明，并声明类型名称而不是函数或对象。它可以在同一行上声明一个或多个标识符（例如int和一个指向int的指针），它可以声明数组和函数类型，指针和引用，类类型等。在这个声明中引入的每个标识符都变成一个类型名称，如果关键字typedef被删除后，它就会变成对象或函数类型的代名词。

除了[类型说明符](#类型说明符)外，typedef说明符不能与任何其他说明符结合使用。它本身是一种存储类的关键字，与auto、extern、mutable、static、register等关键字不能出现在同一个表达式中。

说明符typedef定义的类型名称是现有的类型的别名，并不是新类型的声明。Typedef不能被用于更改现有类型名称的意义（包括typedef定义的名称）。一旦被声明，为了再次引用同一类型，类型名称只能重新声明。Typedef名称仅在它们可见的范围内有效：不同的函数或类声明可以定义具有不同含义的相同名称类型。

注意：

typedef说明符不会出现在不包含声明符的声明中。（C++17后）

|  |
| --- |
| typedef struct X{}; // 不合语法 |

2. 关键字

typedef

3. 注意

类型别名使用不同的语法提供了和typedef相同的功能，并且也能够应用到模板上。

4. 例子

|  |
| --- |
| // 简单typedef  typedef unsigned long ulong;  // 下面的两个对象具有相同的类型  unsigned long l1;  ulong l2;  // 更复杂的typedef  typedef int int\_t, \*intp\_t, (&fp)(int, ulong), arr\_t[10];  //下面的两个对象具有相同的类型  int a1[10];  arr\_t a2;  // 常见的C语法避免使用"struct S"这种方式  typedef struct {int a; int b;} S, \*pS;  //下面两个对象具有相同类型  pS ps1;  S\* ps2;    // 错误: 存储类说明符不能出现在typedef声明中  // typedef static unsigned int uint;    // typedef可以出现在decl-specifier-seq中的任何地方  long unsigned typedef int long ullong;    // std::add\_const, like many other metafunctions, use member typedefs  template< class T>  struct add\_const {  typedef const T type;  };    typedef struct Node {  struct listNode\* next; // declares a new (incomplete) struct type named listNode  } listNode; // error: conflicts with the previously declared struct name |

### 5.13.2 类型别名声明(C++11)

类型别名是一个引用先前定义好的类型的名称（类似于[typedef](#_5.13.1_typedef说明符)）。

别名模板是引用一族类型的名称。

1. 语法

别名声明使用下面的语法：

|  |  |
| --- | --- |
| using identifier attr(可选) = type-id ; | （1） |
| template <template-parameter-list>  using identifier attr(可选) = type-id ; | （2） |

* attr：任意数量的属性的序列
* identifier：这个声明引入的名称，它会变为一个类型名称（1）或者模板名称
* template-parameter-list：参数模板列表
* type-id：抽象声明符或任何其它合法的type-id（可能会引入新的类型，请参考type-id）type-id不能直接或间接引用标识符。 请注意，标识符的声明点位于type-id后面的分号处。

2. 解释

1） 类型别名声明引入了一个名称，该名称可以用作type-id表示的类型的同义词。它不引入新类型，也不能更改现有类型名称的含义。类型别名声明和typedef声明之间没有区别。此声明可能出现在块作用域，类作用域或命名空间作用域。

2） 别名模板是一个模板，当特化时，就等同于替换别名模板中的模板参数为type-id中的模板参数。

|  |
| --- |
| template<class T>  struct Alloc { };  template<class T>  using Vec = vector<T, Alloc<T>>; // type-id是vector<T, Alloc<T>>  Vec<int> v; // Vec<int>与vector<int, Alloc<int>>相同 |

特例化别名模板时生成的类型不允许直接或间接使用自己的类型：

|  |
| --- |
| template<class T>  struct A;  template<class T>  using B = typename A<T>::U; // type-id 是A<T>::U  template<class T>  struct A { typedef B<T> U; };  B<short> b; // error: B<short> 通过A<short>::U使用了自己的类型 |

3. 示例

|  |
| --- |
| #include <string>  #include <ios>  #include <type\_traits>    // type alias, identical to  // typedef std::ios\_base::fmtflags flags;  using flags = std::ios\_base::fmtflags;  // the name 'flags' now denotes a type:  flags fl = std::ios\_base::dec;    // type alias, identical to  // typedef void (\*func)(int, int);  using func = void (\*) (int, int);  // the name 'func' now denotes a pointer to function:  void example(int, int) {}  func f = example;    // alias template  template<class T>  using ptr = T\*;  // the name 'ptr<T>' is now an alias for pointer to T  ptr<int> x;    // type alias used to hide a template parameter  template<class CharT>  using mystring = std::basic\_string<CharT, std::char\_traits<CharT>>;  mystring<char> str;    // type alias can introduce a member typedef name  template<typename T>  struct Container { using value\_type = T; };  // which can be used in generic programming  template<typename Container>  void g(const Container& c) { typename Container::value\_type n; }    // type alias used to simplify the syntax of std::enable\_if  template<typename T>  using Invoke = typename T::type;  template<typename Condition>  using EnableIf = Invoke<std::enable\_if<Condition::value>>;  template<typename T, typename = EnableIf<std::is\_polymorphic<T>>>  int fpoly\_only(T t) { return 1; }    struct S { virtual ~S() {} };    int main()  {  Container<int> c;  g(c); // Container::value\_type will be int in this function  // fpoly\_only(c); // error: enable\_if prohibits this  S s;  fpoly\_only(s); // okay: enable\_if allows this  } |

## 5.14 详细类型说明符（未完待续）

Elaborated type specifiers may be used to refer to a previously-declared class name (class, struct, or union) or to a previously-declared enum name even if the name was hidden by a non-type declaration. They may also be used to declare new class names.

详细类型说明符可用于引用先前声明的类名称（类，结构或联合）或先前声明的枚举，即使名称被非类型声明隐藏。 它们也可能用于声明新的类名。

## 5.15 属性(C++11)

## 5.16 static\_assert(C++11)

# 6 初始化

## 6.1 默认初始化

## 6.2 值初始化(C++03)

## 6.3 拷贝初始化

## 6.4 Direct initialization

## 6.5 Aggregate initialization

## 6.6 列表初始化(C++11)

## 6.7 引用初始化

绑定一个引用和对象，也就是使引用能够代替对象。

### 6.7.1 语法

|  |  |  |
| --- | --- | --- |
| T & ref = object ;  T & ref = { arg1, arg2, ... };  T & ref ( object ) ;  T & ref { arg1, arg2, ... } ; | （1） |  |
| T && ref = object ;  T && ref = { arg1, arg2, ... };  T && ref ( object ) ;  T && ref { arg1, arg2, ... } ; | （2） | （C++11后） |
| R fn ( T & arg ); 或 R fn ( T && arg );  fn ( object )  fn ( { arg1, arg2, ... } ) | （3） |  |
| given T & fn () { 或 T && fn () {  return object ; | （4） |  |
| Class::Class(...) : refmember( expr) {...} | （5） |  |

### 6.7.2 解释

对于类型为T的引用可以使用类型为T的对象、函数或者能够隐式转换为类型为T的对象进行初始化。一旦初始化后，引用就不能再被引用其它对象了。

引用在以下几种情况被初始化：

1) 当声明一个[左值引用](#_5.3.2_左值引用)变量且带有初始化时，

2) 当声明一个[右值引用](#_5.3.3_右值引用)变量且带有初始化时，

3) 函数调用时，其参数中带有引用类型，

4) 在函数返回时,返回一个引用类型时，

5) 使用[成员初始化列表](#_9.10_构造函数和成员初始化列表)初始化引用类型的[非静态数据成员](#_9.3.1_非静态数据成员)时，

引用初始化的作用是：

1) 如果初始化是使用大括号括起来的列表-{arg1，arg2，...}，则遵循[列表初始化](#_6.6_列表初始化(C++11))规则。

2) 如果引用是左值引用：

* 如果对象是一个左值表达式，且它的类型是T或者是从T派生的，且是等于或者小于cv限定的，那么引用被绑定到由左值或者其基类子对象标识的对象上。

|  |
| --- |
| double d = 2.0;  double& rd = d; // rd引用d  const double& rcd = d; // rcd引用d  struct A {};  struct B : A {} b;  A& ra = b; // ra 引用b中的A子对象  const A& rca = b; // ra 引用b中的A子对象 |

* 如果对象是一个左值表达式，并且它的类型可以隐式转换为T或从T派生的类型，等于或者小于cv限定的，那么会考虑源类型及其基类的非显式转换函数返回左值引用，并通过重载决策来选择最好的那一个。然后将引用绑定到由转换函数（或其基类子对象）返回的左值标识的对象，

|  |
| --- |
| struct A{ };  struct B : A { operator int&(); };  int& ir = B(); // ir引用B::operator int&的结果 |

3) 如果引用是右值引用或const左值引用：

* 如果对象是非位域消亡值（xvalue），class纯右值，数组纯右值，右值（自C ++ 17以来）或函数左值，且其类型是T或从T派生的，等于或者小于cv限定的，则引用绑定到初始化表达式的值或其基类子对象([after materializing a temporary if necessary](#_4.8.1_implicit_-)) (since C++17)。

|  |
| --- |
| struct A { };  struct B : A { };  extern B f();  const A& rca2 = f(); // 绑定到右值B的A子对象  A&& rra = f(); // 和上面相同  int i2 = 42;  int&& rri = static\_cast<int&&>(i2); // 直接绑定到i2 |

* If object is a class type expression that is not same or derived from T and (since C++17) that can be implicitly converted to an xvalue, a class prvalue, (until C++17)rvalue (since C++17) or a function value of type that is either T or derived from T, equally or less cv-qualified, then the reference is bound to the result of the conversion or to its base subobject (after materializing a temporary if necessary). (since C++17) User-defined conversions are not considered (since C++17)
* 如果object是一个类型表达式，它与T和（从C ++ 17开始）可以隐式地转换为xvalue，类prvalue（直到C ++ 17）rvalue（自C ++ 17 ）或者是T或从T派生的类型的函数值（等于或小于cv限定），则引用绑定到转换结果或其基本子对象（在实现临时（如果需要）之后）。 （自C ++ 17以来）不考虑用户定义的转换（自C ++ 17以来）

|  |
| --- |
| struct A { };  struct B : A { };  struct X { operator B(); } x;  const A& r = x; // bound to the A subobject of the result of the conversion  B&& rrb = x; // bound directly to the result of the conversion |

* Otherwise, a temporary of type T is constructed and copy-initialized from object. Copy-initialization rules apply (explicit constructors are not considered). If T or the type of object is a class type, the reference is then direct-initialized with the result of the conversion, except user-defined conversions are ignored for that direct-initialization. For non-class types, the reference is bound directly to this temporary (until C++17)the result of the conversion (after materializing a temporary if necessary) (since C++17)
* 否则，构造一个类型T的临时对象并从对象复制初始化。 复制初始化规则适用（不考虑显式构造函数）。 如果T或对象的类型是类类型，则引用将随转换结果一起直接初始化，但对于该直接初始化，用户定义的转换将被忽略。 对于非类的类型，引用直接绑定到这个临时（直到C ++ 17）转换的结果（在实现了临时的必要之后）（自C ++ 17以来）

|  |
| --- |
| const std::string& rs = "abc"; // rs refers to temporary copy-initialized from char array  const double& rcd2 = 2; // rcd2 refers to temporary with value 2.0  int i3 = 2;  double&& rrd3 = i3; // rrd3 refers to temporary with value 2.0 |

### 6.7.3 临时对象的生命周期

每当引用被绑定到临时对象或其子对象时，临时对象的生命周期就会延长以与引用的生命周期相匹配，但以下情况除外：

* 在返回语句中，函数的返回值的临时绑定不会被延长：它会在返回语句结束后被立即销毁。这样的函数总是返回一个[悬空引用](#_5.3.5_悬空引用)。
* 构造函数的初始化列表中，引用成员的临时绑定仅在构造函数退出之前持续存在，并不会和对象生命周期一样长。（注意：这种初始化在DR1696中是不规范的）（直到C++14）
* 函数调用中，引用参数的临时绑定，直到包含该函数调用的整个表达式结束为止：如果该函数返回生命周期超过整个表达式的引用，它将成为[悬空引用](#_5.3.5_悬空引用)。
* 在new表达式中使用的初始化器中引用的临时绑定存在，直到包含new表达式的整个表达式结束，而不是与初始化对象相同。如果初始化对象生命周期超出整个表达式，则其引用成员变成悬空引用。

一般而言，临时对象的生命周期不能通过“传递”而进一步延长：第二次引用，被临时对象绑定的引用所初始化，不影响其生命周期。

### 6.7.4 注意

引用仅在函数参数声明中，函数返回类型声明中，类成员声明中以及extern说明符出现时不带初始化符。

### 6.7.5 例子

|  |
| --- |
| #include <utility>  #include <sstream>  struct S {  int mi;  const std::pair<int,int>& mp; // 引用成员  };  void foo(int) {}  struct A{};  struct B:A{  int n;  operator int&(){ return n; };  };  B bar() {return B();}  //int& bad\_r; // 错误：没有初始化  extern int& ext\_r; // OK  int main()  {  // 左值  int n = 1;  int& r1 = n; // 对象n的左值引用  const int& cr(n); // 引用可以更符合CV限定  volatile int& cv{n}; // 任何初始化语法都可以使用  int& r2 = r1; // 对象n的另一个左值引用  //int& bad = cr; // 错误：缺少CV限定  int& r3 = const\_cast<int&>(cr); // const\_cast被使用，去掉常量性  void (&rf)(int) = foo; // 函数的左值引用  int ar[3];  int (&ra)[3] = ar; // 数组的左值引用  B b;  A& base\_ref = b; // 基类子对象的引用  int& converted\_ref = b; // 转换函数结果的引用  // 右值  //int& bad = 1; // 错误：不能对右值使用左值引用，1为右值  const int& cref = 1; // 绑定到右值  int&& rref = 1; // 绑定到右值    const A& cref2 = bar(); // B的临时对象的子对象A的引用  A&& rref2 = bar(); // 与上一条相同    int&& xref = static\_cast<int&&>(n); // 直接绑定到对象n  //int&& copy\_ref = n; // 错误：不能绑定到左值  double&& copy\_ref = n; // 绑定到一个右值临时对象，它的值是1.0  // 临时对象生命周期的限制  std::ostream& buf\_ref = std::ostringstream() << 'a'; // ostringstream临时对象被绑定到operator<<  // 的左边算子上，但是它的生命周期持续到分号“;”，buf\_ref现在是悬空引用    S a { 1, {2,3} }; // temporary pair {2,3} bound to the reference member  // a.mp and its lifetime is extended to match a  // (Note: does not compile in C++17)  S\* p = new S{ 1, {2,3} }; // temporary pair {2,3} bound to the reference  // member p->mp, but its lifetime ended at the semicolon  // p->mp is a dangling reference  delete p;  } |

## 6.8 Static non-local initialization

## 6.9 zero - constant

## 6.10 Dynamic non-local initialization

## 6.11 ordered - unordered

# 7 函数

## 7.1 函数声明

函数声明引入了函数名称和它的类型。函数定义就是把函数名称、定义和函数体结合起来。

### 7.2.1 函数声明

函数声明可以在任何作用域内出现。出现在类作用域了就是一个成员函数，除非使用了friend修饰符，详情可以参考[成员函数](#_9.3.1_非静态数据成员)和[友邻函数](#_9.7_成员访问-友邻)。

声明函数由返回类型（由声明语法的decl-specifier-seq提供）和函数声明符组成。

|  |  |  |
| --- | --- | --- |
| noptr-declarator ( parameter-list ) cv(可选) ref(可选) except(可选) attr(可选) | （1） |  |
| noptr-declarator ( parameter-list ) cv(可选) ref(可选) except(可选) attr(可选)->trailing | （2） | C++11之后 |

（1）常规函数声明符的语法

（2）拖尾返回类型声明：仅在最外层的函数声明符的后面允许拖尾返回类型。这种情况下，decl-specifier-seq中必须包含关键字auto。

noptr-declarator：任何合法的声明符。如果以\*，&或&&开始，必须用括号括起来。

parameter-list：参数列表。任何空的，逗号分隔的参数列表都可以。

cv(可选)：const/volatile限定符，只允许出现在非静态成员的声明中。

ref(可选)：ref限定符，只允许出现在非静态成员的声明中。

except(可选)：动态异常修饰符（C++17之前）或非异常修饰符（C++11）。

Trailing：拖尾返回类型，如果返回类型依赖于参数名称，比如template <class T, class U> auto add(T t, U u) -> decltype(t + u); 或者很复杂，比如auto fpif(int)->int(\*)(int)。

函数声明符可以和其它的声明符混在一起使用，其中deck-specifier-seq允许：

|  |
| --- |
| // 声明了一个int，一个int\*指针，一个函数，一个函数指针  int a = 1, \*p = NULL, f(), (\*pf)(double);  // decl-specifier-seq是int，声明符f()声明了一个没有参数且返回类型为int的函数（但是没有定义）    struct S  {  virtual int f(char) const, g(int) &&; // 声明2个非静态成员函数  virtual int f(char), x; // 编译错误：关键字virtual只允许出现在非静态成员的声明中  }; |

函数的返回类型不能是函数类型或数组类型（但可以是对它们的指针或引用）。

### 7.2.2 返回类型推导（未完待续）

如果函数声明的decl-specifier-seq包含关键字auto，则可以省略拖尾返回类型，编译器将根据return语句中的表达式的类型推导出拖尾返回类型。如果返回类型不使用decltype（auto），则推导遵循[模板参数推导](#_10.6_模板参数推导)的规则。

|  |
| --- |
| int x = 1;  auto f() { return x; } // 返回类型是int  const auto& f() { return x; } // 返回类型是const int& |

如果返回类型是decltype(auto)，则返回类型与在return语句中使用的表达式被包装在decltype中时获得的类型相同。

|  |
| --- |
| int x = 1;  decltype(auto) f() { return x; } // 返回类型是int，与decltype(x)相同  decltype(auto) f() { return(x); } // 返回类型是int，与decltype((x)) 相同 |

如果有多个return语句，则所有的推导结果必须具有相同类型

|  |
| --- |
| auto f(bool val)  {  if (val) return 123; // 推导返回类型int  else return 3.14f; // 错误：推导返回类型float  } |

如果没有return语句或者return语句的参数是void表达式，则声明的返回类型必须是decltype（auto），在这种情况下推导的返回类型为void，或者（可能是cv-qualified）auto， 在这种情况下，推导出的返回类型是（相同的cv-qualified）void。

|  |
| --- |
| auto f() {} // 返回void  auto g() { return f(); } // 返回void  auto\* x() {} // 错误：不能从void推导出auto\* |

一旦在函数中看到了return语句，从该语句推导出的返回类型可以在函数的其余部分中使用，包括在其他return语句中。

|  |
| --- |
| auto sum(int i)  {  if (i == 1)  return i; // sum的返回类型是int  else  return sum(i - 1) + i; // 正确：sum的返回类型已经知道  } |

如果return语句使用括号初始化列表（brace-init-list），则不允许推导：

|  |
| --- |
| auto func () { return {1, 2, 3}; } // 错误 |

虚函数不能使用返回类型推导：

|  |
| --- |
| struct F  {  virtual auto f() { return 2; } // 错误  }; |

### 7.2.3 参数列表（未完待续）

### 7.2.4 函数定义（未完待续）

## 7.2 默认参数

允许可以在不提供任何默认参数的情况下，调用函数。

其函数声明语法为：

|  |  |
| --- | --- |
| attr(optional) decl-specifier-seq declarator = initializer | （1） |
| attr(optional) decl-specifier-seq abstract-declarator = initializer | （2） |

比如说下面的代码：

|  |
| --- |
| void point(int x = 3, int y = 4);    point(1,2); // calls point(1,2)  point(1); // calls point(1,4)  point(); // calls point(3,4) |

函数声明中，默认参数的后面参数也必须提供默认参数，或者，由相同作用范围内的函数提供。比如说下面的代码：

|  |
| --- |
| int x(int = 1, int); // 错误，假设x()函数之前没有声明    void f(int n, int k = 1);  void f(int n = 0, int k); // 正确：k由相同作用范围内的之前声明函数提供    void g(int, int = 7);  void h() {  void g(int = 1, int); // 错误：不同的作用范围  } |

省略号“...”不是参数，所以跟在默认参数后面。

只允许在函数声明的参数列表中使用默认参数（C++14之后，lambda表达式也可以），函数指针，函数引用，typedef类型声明中都不可以出现默认参数。对于默认模板参数使用相同的规则。

对于非模板函数，如果一个函数开始没有定义默认参数，可以在相同的作用域内，重新声明时添加默认参数。在函数调用的时候，默认参数是该函数所有可见的声明中提供的默认参数的共同体（union）。

|  |
| --- |
| void f(int, int); // #1  void f(int, int = 7); // #2 OK: 添加一个默认参数    void h() {  f(3); // 调用f(3,7)  void f(int =1, int); // 错误: 内部作用域不能通过重新声明添加默认参数  }    void m() { // 新的作用域开始  void f(int, int); // 内部作用域内声明开始，没有默认参数  f(4); // 错误：没有足够的实参调用函数f(int, int)  void f(int, int = 6);  f(4); // OK: 调用f(4,6);  void f(int, int = 6); // 错误:不能在相同的作用域内重新声明一个默认参数  }    void f(int = 1, int); // #3 OK, 添加一个默认参数给 #2    void n() { // 新的作用域开始  f(); // #1, #2, 和#3 调用f(1, 7);  } |

## 7.3 可变参数

## 7.4 Lambda表达式(C++11)

构造闭包：一个未命名的函数对象，能够捕获范围内的变量。

1语法

|  |  |
| --- | --- |
| [捕获] <tparams>(可选)(c++20) (params) specifiers exception attr -> ret requires(可选)(c++20) { body } | （1） |
| [捕获] (参数列表) -> ret { body } | （2） |
| [捕获] (参数列表) { body } | （3） |
| [捕获] { body } | （4） |

（1）完整声明；

（2）声明一个常量lambda表达式：通过复制捕获的对象在lambda体中是const的。

（3）省略返回类型：返回类型可以从return语句中推导出，就像返回值类型声明为auto的函数一样：

（4）省略参数列表：函数不带任何参数，就行普通函数中的无参函数一样。

2 解释

（1）捕获：捕获列表，使用逗号“，”分割，可以选择默认捕获。捕获列表可以像下面这样设置：

* [a, &b]：a通过复制捕获，b通过引用捕获
* [this]：通过引用捕获当前对象
* [&]：通过引用捕获所有lambda表达式里使用的自动存储的变量，以及捕获当前对象，如果存在
* [=]：通过复制捕获所有lambda表达式里使用的自动存储的变量，以及捕获当前对象，如果存在
* []：什么也不捕获

Lambda表达式也可以不通过捕获而使用一个变量，但是该变量必须具有：

* 是非局部变量，有static或thread local存储周期（在这种情况下，变量不能够被捕获）；
* 是使用常量表达式进行了初始化的引用。

Lambda表达式也可以不用捕获就读取变量的值，但必须具有如下条件：

* 是const non-volatile 整形或枚举型且使用常量表达式进行初始化
* 常量表达式且不能复制构造的。[Structured bindings](#_5.5_结构化绑定（C++17后）)不能被捕获（C++17之后）。

（2）<tparams> （C++20）：略。

（3）params：参数列表

（4）specifiers：修饰符，说明是否可以修改捕获的变量。

* mutable：允许在函数体内修改通过复制（值传递）捕获的参数，且调用它们的非const成员函数。
* constexpr：显式地指定函数调用操作符是一个constexpr函数。当这个说明符不存在时，而又恰好满足所有的constexpr函数要求，函数调用操作符就是constexpr。（C++17之后新添加）

（5）exception：异常设定。

（6）attr：属性修饰符，修饰闭包类型的函数调用操作符的类型。

（7）ret：返回类型，如果不存在，就从函数的return语句推导；如果推导不出，就返回void类型。

（8）requires（C++20）：略

（9）body：函数体

3 示例

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;  bool cmp(int a, int b)  {  return a < b;  }  int main()  {  vector<int> myvec{ 3, 2, 5, 7, 3, 2 };  vector<int> lbvec(myvec);  sort(myvec.begin(), myvec.end(), cmp); // 旧式做法  cout << "predicate function:" << endl;  for (int it : myvec)  cout << it << ' ';  cout << endl;  sort(lbvec.begin(), lbvec.end(), [](int a, int b) -> bool { return a < b; }); // Lambda表达式  cout << "lambda expression:" << endl;  for (int it : lbvec)  cout << it << ' ';  } |

在C++11之前，我们使用STL的sort函数，需要提供一个谓词函数（所谓谓词函数就是一个判断式，一个返回bool值得函数或者仿函数）。如果使用C++11的Lambda表达式，我们只需要传入一个匿名函数即可，方便简洁，而且代码的可读性也比旧式的做法好多了。

下面，我们就重点介绍一下Lambda表达式各项的具体用法。

### 7.4.1 闭包类型的成员

Lambda表达式是一个纯右值表达式，它既不是union也不是aggregate（集合），就是一个未命名的class类型，称之为闭包类型。在最小的块作用域，类作用域或包含lambda的命名空间作用域中被声明（用于ADL） 表达。封闭类型有以下成员：

1. ClosureType：：operator()(params)

|  |  |
| --- | --- |
| ret operator() (params) const {body} | 没有使用mutable |
| ret operator() (params) {body} | 使用了mutable |
| template<template-params>  ret operator()(params) const { body } | （C++14之后）  通用Lambda表达式 |
| template<template-params>  ret operator()(params) { body } | （C++14之后）  通用Lambda表达式，关键字mutable被使用 |

当被调用时，执行lambda表达式的主体。当访问变量时，访问它捕获的复制变量（通过复制捕获的实体），或原始对象（通过引用捕获的实体）。除非在lambda表达式中使用关键字mutable，否则const限定的函数调用操作符和通过复制捕获的对象，在operator()内部是不可修改的。函数调用操作符永远不会是volatile限定或virtual限定的。

如果满足常量表达式函数的要求，函数调用操作符总是constexpr的。如果在Lambda表达式中使用了constexpr，那么它也是constexpr的。

对于在params中指定的每一个参数，它的类型被指定为auto，那么，按照参数的先后顺序，一个虚构的模板参数被添加到template-params中。如果params的相应函数成员是函数参数包，那么这个虚构的模板参数可能是一个参数包。

|  |
| --- |
| // 通用lambda, operator()是一个具有2个参数的模板  auto glambda = [](auto a, auto&& b) { return a < b； }；  bool b = glambda(4, 3.14)； // ok  // 通用Lambda, operator()是一个具有1个参数的模板  auto vglambda = [](auto printer)  {  return [=](auto&&... ts) // 通用Lambda表达式, ts参数包  {  printer(std：：forward<decltype(ts)>(ts)...)；  return [=] { printer(ts...)； }； // nullary lambda (takes no parameters)  }；  }；  auto p = vglambda([](auto v1, auto v2, auto v3)  { std：：cout << v1 << v2 << v3； })；  auto q = p(1, 'a', 3.14)； // outputs 1a3.14  q()； // outputs 1a3.14 |

闭包类型ClosureType的操作符operator()不能被显式实例化或显式特化。

If the lambda definition uses an explicit template parameter list, that template parameter list is used with operator(). For every parameter in params whose type is specified as auto, an additional invented template parameter is appended to the end of that template parameter list：

|  |  |
| --- | --- |
| // 通用lambda, operator() 是一具有2个参数的模板  auto glambda = []<class T>(T a, auto&& b) { return a < b； }；    // 通用lambda, operator() 是具有参数包的模板  auto f = []<typename ...Ts>(Ts&& ...ts) {  return foo(std：：forward<Ts>(ts)...)；  }； | （C++20之后） |

the exception specification *exception* on the lambda-expression applies to the function-call operator or operator template.

For the purpose of [name lookup](http://en.cppreference.com/w/cpp/language/lookup), determining the type and value of the [this pointer](http://en.cppreference.com/w/cpp/language/this) and for accessing non-static class members, the body of the closure type's function call operator is considered in the context of the lambda-expression.

|  |
| --- |
| struct X {  int x, y；  int operator()(int)；  void f()  {  // the context of the following lambda is the member function X：：f  [=]()->int  {  return operator()(this->x + y)； // X：：operator()(this->x + (\*this).y)  // this has type X\*  }；  }  }； |

ClosureType's operator() cannot be named in a [friend](http://en.cppreference.com/w/cpp/language/friend) declaration.

**Dangling references**

如果通过引用（隐式或显式）捕获非引用实体，并且在实体的生存期结束后调用闭包对象的函数调用操作符，则会发生未定义的行为。C++闭包不会延长捕获的引用对象的生命周期。

同样适用于通过this指针捕获的对象的生命周期。

### 7.4.2 ClosureType::operator ret(\*)(params)()

|  |  |
| --- | --- |
| using F = ret(\*)(params)；  operator F() const； | (capture-less non-generic lambda) |
| using F = ret(\*)(params)；  constexpr operator F() const； | (capture-less non-generic lambda) |
| template<template-params> using fptr\_t = /\*see below\*/；  template<template-params> operator fptr\_t<template-params>() const； | (capture-less generic lambda) |
| template<template-params> using fptr\_t = /\*see below\*/；  template<template-params> operator fptr\_t<template-params>() const； | (capture-less generic lambda) |

This [user-defined conversion function](http://en.cppreference.com/w/cpp/language/cast_operator) is only defined if the capture list of the lambda-expression is empty. It is a public, constexpr (since C++17) non-virtual, non-explicit, const noexcept (since C++14) member function of the closure object.

|  |  |
| --- | --- |
| A generic captureless lambda has user-defined conversion function template with the same invented template parameter list as the function-call operator template. If the return type is empty or auto, it is obtained by return type deduction on the function template specialization, which, in turn, is obtained by [template argument deduction](http://en.cppreference.com/w/cpp/language/template_argument_deduction) for conversion function templates. | （C++14之后） |
| void f1(int (\*)(int)) {}  void f2(char (\*)(int)) {}  void h(int (\*)(int)) {} // #1  void h(char (\*)(int)) {} // #2  auto glambda = [](auto a) { return a； }；  f1(glambda)； // ok  f2(glambda)； // error： not convertible  h(glambda)； // ok： calls #1 since #2 is not convertible    int& (\*fpi)(int\*) = [](auto\* a)->auto& { return \*a； }； // ok |

The value returned by this conversion function is a pointer to a function with C++ [language linkage](http://en.cppreference.com/w/cpp/language/language_linkage) that, when invoked, has the same effect as invoking the closure object's function call operator directly.

|  |  |  |
| --- | --- | --- |
| This function is constexpr if the function call operator (or specialization, for generic lambdas) is constexpr.   |  | | --- | | auto Fwd= [](int(\*fp)(int), auto a){return fp(a)；}；  auto C=[](auto a){return a；}；  static\_assert(Fwd(C,3)==3)；//OK  auto NC=[](auto a){static int s； return a；}；  static\_assert(Fwd(NC,3)==3)； // error： no specialization can be constexpr because of s |   If the closure object's operator() has a non-throwing exception specification, then the pointer returned by this function has the type pointer to noexcept function. | （C++17之后） |

### 7.4.3 Lambda 捕获

因为Lambda表达式就是一个未命名的函数，所以，与参数传递（值传递、引用传递、指针传递）类似，其捕获方式也有值捕获、引用捕获、隐式捕获（默认捕获方式）。

1.捕获的语法：

|  |  |  |
| --- | --- | --- |
| Identifier | 值捕获-也就是值捕获 |  |
| identifier… | 值捕获，标识符列表 |  |
| identifier initializer | 使用初始化器的值捕获 | C++14之后 |
| & identifier | 引用捕获 |  |
| & identifier… | 引用捕获，标识符列表 |  |
| & identifier initializer | 使用初始化器的引用捕获 | C++14之后 |
| This | 当前对象的引用捕获 |  |
| \*this | 当前对象的值捕获 | C++17之后 |

如果默认捕获是引用（&），那么后面的捕获就不能用引用（&）开始。

|  |
| --- |
| struct S2 { void f(int i)； }；  void S2：：f(int i)  {  [&]{}； // OK： 默认引用捕获  [&, i]{}； // OK： 默认引用捕获，i是值捕获  [&, &i] {}； // 错误： 默认捕获是引用时，后面的捕获不能再使用引用（&）。  [&, this] {}； // OK, 等于[&]  [&, this, i]{}； // OK, 等于[&, i]  } |

但是，如果默认捕获是=，那么后面的捕获必须是（&）开始，或\*this（C++17后），或this（C++20后）。

|  |
| --- |
| struct S2 { void f(int i)； }；  void S2：：f(int i)  {  [=]{}； // OK： 默认值捕获  [=, &i]{}； // OK： 默认值捕获，i是引用捕获  [=, \*this]{}； // C++17之前： 错误： 不合法的语法  // C++17之后： OK：值捕获  [=, this] {}； // C++20之前： 错误： this when = is the default  // since C++20： OK, same as [=]  } |

任何捕获只能出现一次：

|  |
| --- |
| struct S2 { void f(int i)； }；  void S2：：f(int i)  {  [i, i] {}； // 错误： i 重复  [this, \*this] {}； // 错误： "this" 重复 (C++17)  } |

只有在块作用域或默认成员初始化器中定义的lambda表达式，才会有默认捕获或者没有初始化的捕获。对于这样的lambda表达式，作用范围被定义为包括最内层嵌套的函数及其参数。

在没有初始化的情况下（除了this捕获），任何捕获中的标识符都会在lambda范围内使用普通的非限定名称查找来查找。查找的结果必须是在作用范围内声明的具有自动存储周期的变量。变量（或this）被明确地捕获。

|  |  |  |
| --- | --- | --- |
| 使用初始化器的捕获，表现行为就像声明并显式捕获一个声明类型为auto的变量，它的声明域是lambda表达式的主体（也就是说，它不在初始化器的范围之内），除了：   * 如果捕获是通过复制实现，闭包对象的非静态数据成员就是引用那个自动变量的另一种方法； * 如果捕获是通过引用，那么当闭包对象的生命周期结束时，引用变量的生命周期也就结束了。   这常用来捕获如x = std：：move(x)move-only类型   |  | | --- | | int x = 4；  auto y = [&r = x, x = x + 1]()->int  {  r += 2；  return x \* x；  }()； // 更新：：x为6，且初始化y为25。 | | C++14后 |

### 7.4.4 修改捕获变量

前面我们提到过，在Lambda表达式中，如果以传值方式捕获外部变量，则函数体中不能修改该外部变量，否则会引发编译错误。那么有没有办法可以修改值捕获的外部变量呢？这是就需要使用mutable关键字，该关键字用以说明表达式体内的代码可以修改值捕获的变量，示例：

|  |
| --- |
| int main()  {  int a = 123;  auto f = [a]()mutable { cout << ++a; }; // 不会报错  cout << a << endl; // 输出：123  f(); // 输出：124  } |

### 7.4.5 Lambda的参数

Lambda表达式的参数和普通函数的参数类似，那么这里为什么还要拿出来说一下呢？原因是在Lambda表达式中传递参数还有一些限制，主要有以下几点：

1. 参数列表中不能有默认参数；
2. 不支持可变参数；
3. 所有参数必须有参数名；

示例：

|  |
| --- |
| #include <iostream>  #include <functional>  void print\_num(int x)  {  std::cout << " num = " << x << std::endl;  }  int main()  {  int m = [](int x) { return [](int y) { return y \* 2; }(x)+6; }(5);  std::cout << "m:" << m << std::endl; //输出m:16  std::cout << "n:" << [](int x, int y) { return x + y; }(5, 4) << std::endl; //输出n:9    auto gFunc = [](int x) -> std::function<int(int)> { return [=](int y) { return x + y; }; };  auto lFunc = gFunc(4);  std::cout << lFunc(5) << std::endl;  auto hFunc = [](const std::function<int(int)>& f, int z) { return f(z) + 1; };  auto a = hFunc(gFunc(7), 8);  int a\_1 = 111, b = 222;  auto func = [=, &b]()mutable { a\_1 = 22; b = 333; std::cout << "a\_1:" << a\_1 << " b:" << b << std::endl; };  func();  std::cout << "a:" << a << " b:" << b << std::endl;  a = 333;  auto func2 = [=, &a] { a = 444; std::cout << "a:" << a << " b:" << b << std::endl; };  func2();  auto func3 = [](int x) -> std::function<int(int)> { return [=](int y) { return x + y; }; };  std::function<void(int x)> f\_display\_42 = [](int x) { print\_num(x); };  f\_display\_42(44);  } |

## 7.5 inline内联函数（已完成）

使用inline关键字声明函数是内联的。

完全在class/struct/union中定义的函数，无论是成员函数还是非成员函数（friend），隐式地都是内联函数。

\*C++11之后，使用constexpr常量表达式声明的函数隐式地也是内联（inline）函数。使用delete关键字删除掉的函数隐含地也是内联函数。

\*C++17又添加了内联变量，后续研究。

### 7.5.1 描述

内联函数具有下面的属性：

1. 在程序中，可以有内联函数的多个定义，只要每个定义出现在不同的转译单元就行。所有的定义都是相同的。比如，一个内联函数可以定义在头文件中，然后通过#include指令包含到多个源文件中。
2. 在一个转译单元中，所使用的内联函数的定义必须存在。
3. 具有外部链接属性的内联函数（没有被声明为static）拥有额外的属性：
   1. 必须在每个编译单元里都使用inline关键字声明；
   2. 在每一个编译单元里都具有相同的地址。

### 7.5.2 注意

1. 如果具有外部链接属性的内联函数被定义在不同的编译单元里，其行为未定义。
2. 关键字inline不能在块作用域内，用来声明函数。比如，在另一个函数里声明内联函数是不可以的。
3. 已经声明为非内联的函数不能再重新声明为内联函数。
4. 隐含产生的成员函数和任何声明为默认函数的成员函数都如类定义中的其它函数一样都是内联的。
5. 如果在不同的转译单元中声明内联函数，每个转译单元中的函数默认参数集必须相同。
6. 在C语言中，内联函数可以不必在每个转译单元中都声明为内联（最多一个非内联，一个外部内联），函数的定义也可以不同（但是，函数），且函数内部的局部static变量，在相同函数的不同定义中是不同的。

举例：

|  |
| --- |
| // 头文件  #ifndef EXAMPLE\_H  #define EXAMPLE\_H  // 包含在多个源文件的函数定义必须是内联的  inline int sum(int a, int b)  {  return a + b;  }  #endif    // 源文件 #2  #include "example.h"  int a()  {  return sum(1, 2);  }    // 源文件 #1  #include "example.h"  int b()  {  return sum(3, 4);  } |

## 7.6 参数依赖查找（ADL）

**1. 举例说明**

参数依赖检查，又称为ADL（arguments dependent lookup）或Koenig查找，是在函数调用表达式中，也包括对重载运算符的隐式函数调用中，查找非限定函数名称的一组规则。除了通常的非限定名称查找所考虑的作用域和命名空间之外，这些函数名称还会被在其参数的命名空间中查找。

参数依赖检查使得使用定义在不同的命名空间里的操作符成为可能。例如：

|  |
| --- |
| #include <iostream>  int main()  // 在全局命名空间中没有操作符<<，但是ADL检查std命名空间，因为左边的参数在std中，且在  // 其中能找到std：：operator<<(std：：ostream&, const char\*)  std：：cout << "Test\n"；  // 相同，使用函数调用符号样式  operator<<(std：：cout, "Test\n")；    // 但是,下面这句就会报出"Error： 'endl' is not declared in this namespace"这样的错误  // 因为这对于endl()来说，并不是一个函数调用，所以ADL并不适用  std：：cout << endl；    // OK：这是一个函数调用，ADL检查std命名空间，因为endl的参数在std命名空间里,能够发现std：：endl  endl(std：：cout)；    // 但是, 下面这句就会报出"Error： 'endl' is not declared in this namespace"这样的错误  // 因为子表达式(endl)不是一个函数调用表达式  (endl)(std：：cout)；  } |

**2 详细介绍**

First, the argument-dependent lookup is not considered if the lookup set produced by usual [unqualified lookup](http://en.cppreference.com/w/cpp/language/lookup) contains any of the following：

1) 类成员声明

2) a declaration of a function at block scope (that's not a [using-declaration](http://en.cppreference.com/w/cpp/language/namespace#Using-declarations))

3) any declaration that is not a function or a function template (e.g. a function object or another variable whose name conflicts with the name of the function that's being looked up)

Otherwise, for every argument in a function call expression its type is examined to determine the *associated set of namespaces and classes* that it will add to the lookup.

1) For arguments of fundamental type, the associated set of namespaces and classes is empty

2) For arguments of class type (including union), the set consists of

a) The class itself

b) All of its direct and indirect base classes

c) If the class is a [member of another class](http://en.cppreference.com/w/cpp/language/nested_types), the class of which it is a member

d) The innermost enclosing namespaces in the classes added to the set

3) For arguments whose type is a [class template](http://en.cppreference.com/w/cpp/language/class_template) specialization, in addition to the class rules, the following types are examined and their associated classes and namespaces are added to the set

a) The types of all template arguments provided for type template parameters (skipping non-type template parameters and skipping template template parameters)

b) The namespaces in which any template template arguments are members

c) The classes in which any template template arguments are members (if they happen to be class member templates)

4) For arguments of enumeration type, the namespace in which the enumeration is defined is added to the set. If the enumeration type is a member of a class, that class is added to the set.

5) For arguments of type pointer to T or pointer to an array of T, the type T is examined and its associated set of classes and namespaces is added to the set.

6) For arguments of function type, the function parameter types and the function return type are examined and their associated set of classes and namespaces are added to the set.

7) For arguments of type pointer to member function F of class X, the function parameter types, the function return type, and the class X are examined and their associated set of classes and namespaces are added to the set.

8) For arguments of type pointer to data member T of class X, the member type and the type X are both examined and their associated set of classes and namespaces are added to the set.

9) If the argument is the name or the [address-of expression for a set of overloaded functions](http://en.cppreference.com/w/cpp/language/overloaded_address) (or function templates), every function in the overload set is examined and its associated set of classes and namespaces is added to the set.

a) Additionally, if the set of overloads is named by a template-id (template name with template arguments), all of its type template arguments and template template arguments (but not non-type template arguments) are examined and their associated set of classes and namespaces are added to the set.

If any namespace in the associated set of classes and namespaces is an [inline namespace](http://en.cppreference.com/w/cpp/language/namespace), its enclosing namespace is also added to the set.

If any namespace in the associated set of classes and namespaces directly contains an inline namespace, that inline namespace is added to the set.

After the associated set of classes and namespaces is determined, all declarations found in classes of this set are discarded for the purpose of further ADL processing, except namespace-scoped friend functions and function templates, as stated in point 2 below .

The set of declarations found by ordinary [unqualified lookup](http://en.cppreference.com/w/cpp/language/lookup) and the set of declarations found in all elements of the associated set produced by ADL, are merged, with the following special rules

1) [using-directives](http://en.cppreference.com/w/cpp/language/namespace#Using-directives) in the associated namespaces are ignored

2) namespace-scoped friend functions (and function templates) that are declared in an associated class are visible through ADL even if they are not visible through ordinary lookup

3) all names except for the functions and function templates are ignored (no collision with variables)

**Notes**

Because of argument-dependent lookup, non-member functions and non-member operators defined in the same namespace as a class are considered part of the public interface of that class (if they are found through ADL) [[1]](http://en.cppreference.com/w/cpp/language/adl#cite_note-1).

ADL is the reason behind the established idiom for swapping two objects in generic code：

|  |
| --- |
| using std：：swap；  swap(obj1, obj2)； |

because calling [std：：swap](http://en.cppreference.com/w/cpp/algorithm/swap)(obj1, obj2) directly would not consider the user-defined swap() functions that could be defined in the same namespace as the types of obj1 or obj2, and just calling the unqualified swap(obj1, obj2)would call nothing if no user-defined overload was provided. In particular, [std：：iter\_swap](http://en.cppreference.com/w/cpp/algorithm/iter_swap) and all other standard library algorithms use this approach when dealing with [Swappable](http://en.cppreference.com/w/cpp/concept/Swappable) types.

Name lookup rules make it impractical to declare operators in global or user-defined namespace that operate on types from the std namespace, e.g. a custom operator>> or operator+ for [std：：vector](http://en.cppreference.com/w/cpp/container/vector) or for [std：：pair](http://en.cppreference.com/w/cpp/utility/pair) (unless the element types of the vector/pair are user-defined types, which would add their namespace to ADL). Such operators would not be looked up from template instantiations, such as the standard library algorithms. See [dependent names](http://en.cppreference.com/w/cpp/language/dependent_name)for further details.

ADL can find a [friend function](http://en.cppreference.com/w/cpp/language/friend) (typically, an overloaded operator) that is defined entirely within a class or class template, even if it was never declared at namespace level.

|  |
| --- |
| template<typename T>  struct number  {  number(int)；  friend number gcd(number x, number y) { return 0； }； // definition within  // a class template  }；  // unless a matching declaration is provided gcd is an invisible (except through ADL)  // member of this namespace  void g() {  number<double> a(3), b(4)；  a = gcd(a,b)； // finds gcd because number<double> is an associated class,  // making gcd visible in its namespace (global scope)  // b = gcd(3,4)； // Error； gcd is not visible  } |

Although a function call can be resolved through ADL even if ordinary lookup finds nothing, a function call to a [function template](http://en.cppreference.com/w/cpp/language/function_template) with explicitly-specified template arguments requires that there is a declaration of the template found by ordinary lookup (otherwise, it is a syntax error to encounter an unknown name followed by a less-than character)

|  |
| --- |
| namespace N1 {  struct S {}；  template<int X> void f(S)；  }  namespace N2 {  template<class T> void f(T t)；  }  void g(N1：：S s) {  f<3>(s)； // Syntax error (unqualified lookup finds no f)  N1：：f<3>(s)； // OK, qualified lookup finds the template 'f'  N2：：f<3>(s)； // Error： N2：：f does not take a non-type parameter  // N1：：f is not looked up because ADL only works  // with unqualified names  using N2：：f；  f<3>(s)； // OK： Unqualified lookup now finds N2：：f  // then ADL kicks in because this name is unqualified  // and finds N1：：f  } |

In the following contexts ADL-only lookup (that is, lookup in associated namespaces only) takes place：

* the lookup of non-member functions begin and end performed by the [range-for](http://en.cppreference.com/w/cpp/language/range-for) loop if member lookup fails
* the [dependent name lookup](http://en.cppreference.com/w/cpp/language/dependent_name#Lookup_rules) from the point of template instantiation.

|  |
| --- |
| * the lookup of non-member function get performed by [structured binding declaration](http://en.cppreference.com/w/cpp/language/structured_binding) for tuple-like types (since C++17) |

**Examples**

|  |
| --- |
| namespace A {  struct X；  struct Y；  void f(int)；  void g(X)；  }    namespace B {  void f(int i) {  f(i)； // calls B：：f (endless recursion)  }  void g(A：：X x) {  g(x)； // Error： ambiguous between B：：g (ordinary lookup)  // and A：：g (argument-dependent lookup)  }  void h(A：：Y y) {  h(y)； // calls B：：h (endless recursion)： ADL examines the A namespace  // but finds no A：：h, so only B：：h from ordinary lookup is used  }  } |

## 7.7 重载决策

为了编译函数调用，编译器必须首先执行[名称查找](#_1.7_名称查找)，对于函数来说，可能会涉及[参数依赖查找](#_7.6_参数依赖查找（ADL）)，而对于函数模板，可能会跟随着[模板参数推导](#_10.5_模板参数推导)。如果这些步骤产生多个候选函数，则执行重载决策来选择将实际调用的函数。

通常，参数最匹配的函数是被调用的函数。

对于其中可能出现重载函数名称的其它上下文，请参阅[获取重载函数的地址](#_7.9_重载集的地址)。

1. 详细内容

Before overload resolution begins, the functions selected by name lookup and template argument deduction are combined to form the set of *candidate functions* (the exact criteria depend on the context in which overload resolution takes place, see below).

If any candidate function is a [member function](http://en.cppreference.com/w/cpp/language/member_functions) (static or non-static), but not a constructor, it is treated as if it has an extra parameter (*implicit object parameter*) which represents the object for which they are called and appears before the first of the actual parameters.

Similarly, the object on which a member function is being called is prepended to the argument list as the *implied object argument*.

For member functions of class X, the type of the implicit object parameter is affected by cv-qualifications and ref-qualifications of the member function as described in [member functions](http://en.cppreference.com/w/cpp/language/member_functions).

The user-defined conversion functions are considered to be members of the *implied object argument* for the purpose of determining the type of the *implicit object parameter*.

The member functions introduced by a using-declaration into a derived class are considered to be members of the derived class for the purpose of defining the type of the *implicit* object parameter*.*

For the static member functions, the *implicit object parameter* is considered to match any object： its type is not examined and no conversion sequence is attempted for it.

For the rest of overload resolution, the *implied object argument* is indistinguishable from other arguments, but the following special rules apply to the *implicit object parameter*：

1) user-defined conversions cannot be applied to the implicit object parameter

2) rvalues can be bound to non-const implicit object parameter (unless this is for a ref-qualified member function)(since C++11) and do not affect the ranking of the implicit conversions.

|  |
| --- |
| struct B { void f(int)； }；  struct A { operator B&()； }；  A a；  a.B：：f(1)； // Error： user-defined conversions cannot be applied  // to the implicit object parameter  static\_cast<B&>(a).f(1)； // OK |

**Candidate functions**

The set of candidate functions and the list of arguments is prepared in a unique way for each of the contexts where overload resolution is used：

**Call to a named function**

If E in a [function call expression](http://en.cppreference.com/w/cpp/language/operator_other#Built-in_function_call_operator) E(args) names a set of overloaded functions and/or function templates (but not callable objects), the following rules are followed：

* If the expression E has the form PA->B or A.B (where A has class type cv T), then B is [looked up](http://en.cppreference.com/w/cpp/language/lookup) as a member function of T. The function declarations found by that lookup are the candidate functions. The argument list for the purpose of overload resolution has the implied object argument of type cv T.
* If the expression E is a [primary expression](http://en.cppreference.com/w/cpp/language/expressions#Primary_expressions), the name is [looked up](http://en.cppreference.com/w/cpp/language/lookup) following normal rules for function calls (which may involve [ADL](http://en.cppreference.com/w/cpp/language/adl)). The function declarations found by this lookup are (due to the way lookup works) either：

a) all non-member functions (in which case the argument list for the purpose of overload resolution is exactly the argument list used in the function call expression)

b) all member functions of some class T, in which case, if [this](http://en.cppreference.com/w/cpp/language/this) is in scope and refers to T, \*this is used as the implied object argument. Otherwise (if this is not in scope or does not point to T, a fake object of type T is used as the implied object argument, and if overload resolution subsequently selects a non-static member function, the program is ill-formed.

**Call to a class object**

If E in a [function call expression](http://en.cppreference.com/w/cpp/language/operator_other#Built-in_function_call_operator) E(args) has class type cv T, then

* The function-call operators of T are obtained by ordinary [lookup](http://en.cppreference.com/w/cpp/language/lookup) of the name operator() in the context of the expression (E).operator(), and every declaration found is added to the set of candidate functions.
* For each non-explicit [user-defined conversion function](http://en.cppreference.com/w/cpp/language/cast_operator) in T or in a base of T (unless hidden), whose cv-qualifiers is same or greater than T's cv-qualifiers, and where the conversion function converts to：
* to pointer-to-function
* to reference-to-pointer-to-function
* to reference-to-function

then a *surrogate call function* with a unique name whose first parameter is the result of the conversion, the remaining parameters are the parameter-list accepted by the result of the conversion, and the return type is the return type of the result of the conversion, is added to the set of candidate functions. If this surrogate function is selected by the subsequent overload resolution, then the user-defined conversion function will be called and then the result of the conversion will be called.

In any case, the argument list for the purpose of overload resolution is the argument list of the function call expression preceded by the implied object argument E (when matching against the surrogate function, the user-defined conversion will automatically convert the implied object argument to the first argument of the surrogate function).

|  |
| --- |
| int f1(int)；  int f2(float)；  struct A {  using fp1 = int(\*)(int)；  operator fp1() { return f1； } // conversion function to pointer to function  using fp2 = int(\*)(float)；  operator fp2() { return f2； } // conversion function to pointer to function  } a；  int i = a(1)； // calls f1 via pointer returned from conversion function |

**Call to an overloaded operator**

If at least one of the arguments to an operator in an expression has a class type or an enumeration type, both [builtin operators](http://en.cppreference.com/w/cpp/language/expressions#Operators) and [user-defined operator overloads](http://en.cppreference.com/w/cpp/language/operators) participate in overload resolution, with the set of candidate functions selected as follows：

For a unary operator **@** whose argument has type T1 (after removing cv-qualifications), or binary operator **@** whose left operand has type T1 and right operand of type T2 (after removing cv-qualifications), three sets of candidate functions are prepared：

1) *member candidates*： if T1 is a complete class or a class currently being defined, the set of member candidates is the result of [qualified name lookup](http://en.cppreference.com/w/cpp/language/lookup) of T1：：operator@. In all other cases, the set of member candidates is empty.

2) *non-member candidates*： For the operators where [operator overloading](http://en.cppreference.com/w/cpp/language/operators) permits non-member forms, all declarations found by [unqualified name lookup](http://en.cppreference.com/w/cpp/language/lookup) of operator@ in the context of the expression (which may involve [ADL](http://en.cppreference.com/w/cpp/language/adl)), except that member function declarations are ignored and do not prevent the lookup from continuing into the next enclosing scope. If both operands of a binary operator or the only operand of a unary operator has enumeration type, the only functions from the lookup set that become non-member candidates are the ones whose parameter has that enumeration type (or reference to that enumeration type)

3) *built-in candidates*： For operator,, the unary operator&, and the operator->, the set of built-in candidates is empty. For other operators built-in candidates are the ones listed in [built-in operator pages](http://en.cppreference.com/w/cpp/language/expressions#Operators) as long as all operands can be implicitly converted to their parameters. If any built-in candidate has the same parameter list as a non-member candidate that isn't a function template specialization, it is not added to the list of built-in candidates. When the built-in assignment operators are considered, the conversions from their left-hand arguments are restricted： user-defined conversions are not considered.

The set of candidate functions to be submitted for overload resolution is a union of the three sets above. The argument list for the purpose of overload resolution consists of the operands of the operator except for operator->, where the second operand is not an argument for the function call (see [member access operator](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_member_access_operators)).

|  |
| --- |
| struct A {  operator int()； // user-defined conversion  }；  A operator+(const A&, const A&)； // non-member user-defined operator  void m()  {  A a, b；  a + b； // member-candidates： none  // non-member candidates： operator+(a,b)  // built-in candidates： int(a) + int(b)  // overload resolution chooses operator+(a,b)  } |

If the overload resolution selects a built-in candidate, the [user-defined conversion sequence](http://en.cppreference.com/w/cpp/language/implicit_cast) from an operand of class type is not allowed to have a second standard conversion sequence： the user-defined conversion function must give the expected operand type directly：

|  |
| --- |
| struct Y { operator int\*()； }； // Y is convertible to int\*  int \*a = Y() + 100.0； // error： no operator+ between pointer and double |

For operator,, the unary operator&, and operator->, if there are no viable functions (see below) in the set of candidate functions, then the operator is reinterpreted as a built-in.

**Initialization by constructor**

When an object of class type is [direct-initialized](http://en.cppreference.com/w/cpp/language/direct_initialization) or [default-initialized](http://en.cppreference.com/w/cpp/language/default_initialization) outside a [copy-initialization](http://en.cppreference.com/w/cpp/language/copy_initialization) context, the candidate functions are all constructors of the class being initialized. The argument list is the expression list of the initializer.

When an object of class type is copy-initialized from an object of the same or derived class type, or default-initialized in a copy-initialization context, the candidate functions are all [converting constructors](http://en.cppreference.com/w/cpp/language/converting_constructor) of the class being initialized. The argument list is the expression of the initializer.

**Copy-initialization by conversion**

If [copy-initialization](http://en.cppreference.com/w/cpp/language/copy_initialization) of an object of class type requires that a [user-defined conversion function](http://en.cppreference.com/w/cpp/language/cast_operator) is called to convert the initializer expression of type cv S to the type cv T of the object being initialized, the following functions are candidate functions：

* all [converting constructors](http://en.cppreference.com/w/cpp/language/converting_constructor) of T
* the non-explicit conversion functions from S and its base classes (unless hidden) to T or class derived from T or a reference to such. If this copy-initialization is part of the direct-initialization sequence of *cv* T (initializing a reference to be bound to the first parameter of a constructor that takes a reference to cv T), then explicit conversion functions are also considered.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the first argument of the constructor or against the implicit object argument of the conversion function.

**Non-class initialization by conversion**

When initialization of an object of non-class type cv1 T requires a [user-defined conversion function](http://en.cppreference.com/w/cpp/language/cast_operator) to convert from an initializer expression of class type cv S, the following functions are candidates：

* the non-explicit user-defined conversion functions of S and its base classes (unless hidden) that produce type T or a type convertible to T by a [standard conversion sequence](http://en.cppreference.com/w/cpp/language/implicit_cast), or a reference to such type. cv qualifiers on the returned type are ignored for the purpose of selecting candidate functions.
* if this is [direct-initialization](http://en.cppreference.com/w/cpp/language/direct_initialization), the explicit user-defined conversion functions of S and its base classes (unless hidden) that produce type T or a type convertible to T by a [qualification conversion](http://en.cppreference.com/w/cpp/language/implicit_cast), or a reference to such type, are also considered.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the implicit object argument of the conversion function.

**Reference initialization by conversion**

During [reference initialization](http://en.cppreference.com/w/cpp/language/reference_initialization), where the reference to cv1 T is bound to the glvalue or class prvalue result of a conversion from the initializer expression from the class type cv2 S, the following functions are selected for the candidate set：

* the non-explicit user-defined conversion functions of S and its base classes (unless hidden) to the type
* (when initializing lvalue reference or rvalue reference to function) lvalue reference to cv2 T2
* (when initializing rvalue reference or lvalue reference to function) cv2 T2 or rvalue reference to cv2 T2

where cv2 T2 is reference-compatible with cv1 T

* for direct initializaton, the explicit user-defined conversion functions are also considered if T2 is the same type as T or can be converted to type T with a qualification conversion.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the implicit object argument of the conversion function.

**List-initialization**

When an object of non-aggregate class type T is [list-initialized](http://en.cppreference.com/w/cpp/language/list_initialization), two-phase overload resolution takes place.

* at phase 1, the candidate functions are all initializer-list constructors of T and the argument list for the purpose of overload resolution consists of a single initializer list argument
* if overload resolution fails at phase 1, phase 2 is entered, where the candidate functions are all constructors of Tand the argument list for the purpose of overload resolution consists of the individual elements of the initializer list.

If the initializer list is empty and T has a default constructor, phase 1 is skipped.

In copy-list-initialization, if phase 2 selects an explicit constructor, the initialization is ill-formed (as opposed to all over copy-initializations where explicit constructors are not even considered).

**Viable functions**

Given the set of candidate functions, constructed as described above, the next step of overload resolution is examining arguments and parameters to reduce the set to the set of *viable functions*

To be included in the set of viable functions, the candidate function must satisfy the following：

1) If there are M arguments, the candidate function that has exactly M parameters is viable

2) If the candidate function has less than M parameters, but has an [ellipsis parameter](http://en.cppreference.com/w/cpp/language/variadic_arguments), it is viable.

3) If the candidate function has more than M parameters and the M+1'st parameter and all parameters that follow must have default arguments, it is viable. For the rest of overload resolution, the parameter list is truncated at M.

|  |  |
| --- | --- |
| 4) If the function has an associated [constraint](http://en.cppreference.com/w/cpp/language/constraints), it must be satisfied | (since C++20) |

5) For every argument there must be at least one implicit conversion sequence that converts it to the corresponding parameter.

6) If any parameter has reference type, reference binding is accounted for at this step： if an rvalue argument corresponds to non-const lvalue reference parameter or an lvalue argument corresponds to rvalue reference parameter, the function is not viable.

User-defined conversions (both converting constructors and user-defined conversion functions) are prohibited from taking part in implicit conversion sequence where it would make it possible to apply more than one user-defined conversion. Specifically, they are prohibited if the target of the conversion is the first parameter of a constructor or the implicit object parameter of a user-defined conversion function, and that constructor/user-defined conversion is a candidate for

* copy-initialization of a class by user-defined conversion,
* initialization by a conversion function,
* initialization by conversion function for direct reference binding,
* initialization by constructor where the argument is a temporary in class copy-initialization,
* initialization by list-initialization where the initializer list has exactly one element that is itself an initializer list, and the target is the first parameter of a constructor of class X, and the conversion is to X or reference to (possibly cv-qualified) X

|  |
| --- |
| struct A { A(int)； }；  struct B { B(A)； }；  B b{ {0} }； // list-init of B  // candidates： B(const B&), B(B&&), B(A)  // {0} -> B&& not viable： would have to call B(A)  // {0} -> const B&： not viable： would have to bind to rvalue, would have to call B(A)  // {0} -> A viable. Calls A(int)： user-defined conversion to A is not banned |

**Best viable function**

For each pair of viable function F1 and F2, the implicit conversion sequences from the i-th argument to i-th parameter are ranked to determine which one is better (except the first argument, the *implicit object argument* for static member functions has no effect on the ranking)

F1 is determined to be a better function than F2 if implicit conversions for all arguments of F1 are *not worse* than the implicit conversions for all arguments of F2, and

1) there is at least one argument of F1 whose implicit conversion is *better* than the corresponding implicit conversion for that argument of F2

2) or. if not that, (only in context of non-class initialization by conversion), the standard conversion sequence from the return type of F1 to the type being initialized is *better* than the standard conversion sequence from the return type of F2

3) or, if not that, F1 is a non-template function while F2 is a template specialization

4) or, if not that, F1 and F2 are both template specializations and F1 is *more specialized* according to the [partial ordering rules for template specializations](http://en.cppreference.com/w/cpp/language/function_template#Function_template_overloading)

|  |  |
| --- | --- |
| 5) or, if not that, F1 and F2 are non-template functions with the same parameter-type-lists, and F1 is more constrained than F2 according to the [partial ordering of constraints](http://en.cppreference.com/w/cpp/language/constraints) | (since C++20) |

|  |
| --- |
| 6) or, if not that, F1 is generated from a [user-defined deduction-guide](http://en.cppreference.com/w/cpp/language/deduction_guide) and F2 is not  7) or, if not that, F1 is the [copy deduction candidate](http://en.cppreference.com/w/cpp/language/deduction_guide) and F2 is not  8) or, if not that, F1 is generated from a non-template constructor and F2 is generated from a constructor template |

|  |
| --- |
| template <class T> struct A {  using value\_type = T；  A(value\_type)； // #1  A(const A&)； // #2  A(T, T, int)； // #3  template<class U> A(int, T, U)； // #4  }； // #5 is A(A), the copy deduction candidate  A x (1, 2, 3)； // uses #3, generated from a non-template constructor  template <class T> A(T) -> A<T>； // #6, less specialized than #5  A a (42)； // uses #6 to deduce A<int> and #1 to initialize  A b = a； // uses #5 to deduce A<int> and #2 to initialize  template <class T> A(A<T>) -> A<A<T>>； // #7, as specialized as #5  A b2 = a； // uses #7 to deduce A<A<int>> and #1 to initialize |

These pair-wise comparisons are applied to all viable functions. If exactly one viable function is better than all others, overload resolution succeeds and this function is called. Otherwise, compilation fails.

|  |
| --- |
| void Fcn(const int\*, short)； // overload #1  void Fcn(int\*, int)； // overload #2  int i；  short s = 0；  void f() {  Fcn(&i, 1L)； // 1st argument： &i -> int\* is better than &i -> const int\*  // 2nd argument： 1L -> short and 1L -> int are equivalent  // calls Fcn(int\*, int)    Fcn(&i,'c')； // 1st argument： &i -> int\* is better than &i -> const int\*  // 2nd argument： 'c' -> int is better than 'c' -> short  // calls Fcn(int\*, int)    Fcn(&i, s)； // 1st argument： &i -> int\* is better than &i -> const int\*  // 2nd argument： s -> short is better than s -> int  // no winner, compilation error  } |

**Ranking of implicit conversion sequences**

The argument-parameter implicit conversion sequences considered by overload resolution correspond to [implicit conversions](http://en.cppreference.com/w/cpp/language/implicit_cast) used in [copy initialization](http://en.cppreference.com/w/cpp/language/copy_initialization) (for non-reference parameters), except that when considering conversion to the implicit object parameter or to the left-hand side of assignment operator, conversions that create temporary objects are not considered.

Each [type of standard conversion sequence](http://en.cppreference.com/w/cpp/language/implicit_cast) is assigned one of three ranks：

1) **Exact match**： no conversion required, lvalue-to-rvalue conversion, qualification conversion, function pointer conversion, (since C++17) user-defined conversion of class type to the same class

2) **Promotion**： integral promotion, floating-point promotion

3) **Conversion**： integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

The rank of the standard conversion sequence is the worst of the ranks of the standard conversions it holds (there may be up to [three conversions](http://en.cppreference.com/w/cpp/language/implicit_cast))

Binding of a reference parameter directly to the argument expression is either Identity or a derived-to-base Conversion：

|  |
| --- |
| struct Base {}；  struct Derived ： Base {} d；  int f(Base&)； // overload #1  int f(Derived&)； // overload #2  int i = f(d)； // d -> Derived& has rank Exact Match  // d -> Base& has rank Conversion  // calls f(Derived&) |

Since ranking of conversion sequences operates with types and value categories only, a [bit field](http://en.cppreference.com/w/cpp/language/bit_field) can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.

2) A user-defined conversion sequence is always *better* than an [ellipsis conversion](http://en.cppreference.com/w/cpp/language/variadic_arguments) sequence

3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if

a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion

b) Or, if not that, the rank of S1 is better than the rank of S2

c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2binds an lvalue reference to an rvalue

|  |
| --- |
| int i；  int f1()；  int g(const int&)； // overload #1  int g(const int&&)； // overload #2  int j = g(i)； // lvalue int -> const int& is the only valid conversion  int k = g(f1())； // rvalue int -> const int&& better than rvalue int -> const int& |

d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

|  |
| --- |
| int f(void(&)())； // overload #1  int f(void(&&)())； // overload #2  void g()；  int i1 = f(g)； // calls #1 |

e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is less cv-qualified than S2's.

|  |
| --- |
| int f(const int &)； // overload #1  int f(int &)； // overload #2 (both references)  int g(const int &)； // overload #1  int g(int)； // overload #2  int i；  int j = f(i)； // lvalue i -> int& is better than lvalue int -> const int&  // calls f(int&)  int k = g(i)； // lvalue i -> const int& ranks Exact Match  // lvalue i -> rvalue int ranks Exact Match  // ambiguous overload： compilation error |

f) Or, if not that, S1 and S2 only differ in qualification conversion, and the cv-qualification of the result of S1 is a subset of the cv-qualification of the result of S2

|  |
| --- |
| int f(const int\*)；  int f(int\*)；  int i；  int j = f(&i)； // &i -> int\* is better than &i -> const int\*, calls f(int\*) |

4) A user-defined conversion sequence U1 is better than a user-defined conversion sequence U2 if they call the same constructor/user-defined conversion function or initialize the same class with aggregate-initialization, and in either case the second standard conversion sequence in U1 is better than the second standard conversion sequence in U2

|  |
| --- |
| struct A {  operator short()； // user-defined conversion function  } a；  int f(int)； // overload #1  int f(float)； // overload #2  int i = f(a)； // A -> short, followed by short -> int (rank Promotion)  // A -> short, followed by short -> float (rank Conversion)  // calls f(int) |

5) A list-initialization sequence L1 is *better* than list-initialization sequence L2 if L1 initializes an [std：：initializer\_list](http://en.cppreference.com/w/cpp/utility/initializer_list) parameter, while L2 does not.

|  |
| --- |
| void f1(int)； // #1  void f1(std：：initializer\_list<long>)； // #2  void g1() { f1({42})； } // chooses #2    void f2(std：：pair<const char\*, const char\*>)； // #3  void f2(std：：initializer\_list<std：：string>)； // #4  void g2() { f2({"foo","bar"})； } // chooses #4 |

6) A list-initialization sequence L1 is better than list-initialization sequence L2 if the corresponding parameters are references to arrays, and L1 converts to type "array of N1 T," L2 converts to type "array of N2 T", and N1 is smaller than N2.

## 7.8 操作符重载

操作符重载，就是把已经定义的、有一定功能的操作符进行重新定义，来完成更为细致具体的运算等功能。操作符重载可以将概括性的抽象操作符具体化，便于外部调用而无需知晓内部具体运算过程。

### 7.8.1 语法

重载运算符是具有特殊函数名称的函数：

|  |  |  |
| --- | --- | --- |
| operator op | （1） |  |
| operator type | （2） |  |
| operator new  operator new [] | （3） |  |
| operator delete  operator delete [] | （4） |  |
| operator "" *suffix-identifier* | （5） | （C++11后） |

op – 可以是以下操作符的任一个:+ - \* / % ^ & | ~ ! = < > += -= \*= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> (C++20后) && || ++ -- , ->\* -> ( ) [ ]

（1）重载操作符

（2）用户自定义转换函数

（3）内存分配函数（new等）

（4）内存释放函数（delete等）

（5）[用户自定义文字](#_4.9.4_user-defined_(C++11))

### 7.8.2 重载操作符

当运算符出现在表达式中，并且其至少一个操作数具有一个类类型或一个枚举类型时，则使用重载解析来确定调用的用户定义函数，这些函数的特征满足下面的条件：

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 成员函数 | 非成员函数 | 举例 |
| @a | (a).operator@() | operator@(a) | !std::cin调用std::cin.operator!() |
| a@b | (a).operator@ (b) | operator@ (a, b) | std::cout << 42调用std::cout.operator<<(42) |
| a=b | (a).operator= (b) | 不能作为非成员函数 | std::string s; s="abc";调用s.operator=("abc") |
| a(b...) | (a).operator()(b...) | 不能作为非成员函数 | std::random\_device r;auto n=r();调用 r.operator()() |
| a[b] | (a).operator[](b) | 不能作为非成员函数 | std::map<int,int> m;m[1]=2;调用m.operator[](1) |
| a-> | (a).operator-> ( ) | 不能作为非成员函数 | auto p = std::make\_unique<S>(); p->bar() calls p.operator->() |
| a@ | (a).operator@ (0) | operator@ (a, 0) | std::vector<int>::iterator i = v.begin(); i++ 调用 i.operator++(0) |
| 在此表中，@是代表所有匹配运算符的占位符：@a中的所有前缀运算符，除a中的->以外的所有后缀运算符，除a以外的所有中缀运算符 | | | |

注意：为了重载用户定义的转换函数，[用户定义的文字](#_4.9.4_user-defined_(C++11))，分配和释放内存函数请参阅它们各自的文章。

使用函数表示法可以调用重载操作符（但不是内嵌操作符）：

|  |
| --- |
| std::string str = "Hello,";  str.operator+=("world"); // 与str += "world";相等  operator<<(operator<<(std::cout, str) , '\n'); // 与std::cout << str << '\n';相同  // (C++17后) 排序除外 |

### 7.8.3 限制

### 7.8.4 规范

除上述限制之外，该语言对重载操作符所做的操作或返回类型（它不参与重载解析）没有其他限制，但通常，重载操作符的行为应该尽可能与内嵌运算符一致：运算符+期望就是算子相加，而不是乘以其它参数，运算符=期望就是赋值等等。相关的运算符的行为应该类似（运算符+和运算符+=执行相同的加法运算）。返回类型受到使用该操作符的表达式的限制：例如，赋值操作符通过引用返回，以便可以写a = b = c = d，因为内嵌操作符允许这样做。

### 7.8.5 赋值操作符

### 7.8.6 函数调用操作符

### 7.8.7 自增和自减

### 7.8.8 二进制算术运算符

### 7.8.9 关系运算符

### 7.8.10 数组下标操作符

### 7.8.11 位算术运算符

### 7.8.12 布尔运算符

### 7.8.13 很少重载的运算符

### 7.8.14 举例

## 7.9 重载集的地址

Besides [function-call expressions](http://en.cppreference.com/w/cpp/language/operator_other), where [overload resolution](http://en.cppreference.com/w/cpp/language/overload_resolution) takes place, the name of an overloaded function may appear in the following 7 contexts：

除了函数调用表达式（其中发生重载解析）之外，重载函数的名称可能会出现在以下7个上下文中：

1) 对象或引用的初始化中

2) 赋值表达式的右边

3) 作为函数调用参数

4) 作为用户自定义操作符参数

5) return语句

6) [explicit cast](http://en.cppreference.com/w/cpp/language/explicit_cast) 或 [static cast](http://en.cppreference.com/w/cpp/language/static_cast) 参数

7) 无类型 [template argument](http://en.cppreference.com/w/cpp/language/template_parameters)

In each context, the name of an overloaded function may be preceded by address-of operator **&** and may be enclosed in a redundant set of parentheses.

In all these contexts, the function selected from the overload set is the function whose type matches the pointer to function, reference to function, or pointer to member function type that is expected by *target*： the object or reference being initialized, the left-hand side of the assignment, function or operator parameter, the return type of a function, the target type of a cast, or the type of the template parameter, respectively.

The type of the function must match the target exactly, no implicit conversions are considered (e.g. a function returning a pointer to derived won't get selected when initializing a pointer to function returning a pointer to base)

If the function name names a function template, then, first, [template argument deduction](http://en.cppreference.com/w/cpp/language/template_argument_deduction) is done, and if it succeeds, it produces a single template specialization which is added to the set of overloads to consider. All functions whose associated [constraints](http://en.cppreference.com/w/cpp/language/constraints) are not satisfied are dropped from the set. (since C++20) If more than one function from the set matches the target, and at least one function is non-template, the template specializations are eliminated from consideration. For any pair of non-template functions where one is [more constrained](http://en.cppreference.com/w/cpp/language/constraints) than another, the less constrained function is dropped from the set (since C++20). If all remaining candidates are template specializations, [less specialized](http://en.cppreference.com/w/cpp/language/partial_specialization) ones are removed if more specialized are available. If more than one candidate remains after the removals, the program is ill-formed.

# 8 语句

## 8.1 条件语句

8.1.1 if条件语句

8.1.2 switch条件语句

## 8.2 for - range-for(C++11)

## 8.3 while - do-while

## 8.4 continue - break - goto - return

## 8.5 synchronized and atomic(TM TS)

# 9 类

## 9.1 类和共同体（已完成）

### 9.1.1 类声明

class和struct是用户自定义的类型，其声明语法如下：

|  |
| --- |
| class-key attr class-head-name base-clause { member-specification } |

说明如下：

* class-key：关键字class或struct。
* attr(C++11)：属性序列，可选，数量任意，可以包括[alignas说明符](#_5.12_alignas)
* class-head-name：类名称。可以是限定名称，也可以在后面添加[final](#_9.8.2_final修饰符(C++11之后))关键字，从而不被继承。如果这个类不需要命名，还可以省略。（注意的是，未命名类是不能加final限定的）。
* base-clause：一个或多个父类的列表，以及用于每个父类的继承模型（请参考[派生类](#_9.5_派生类_-)）
* member-specification：访问级别修饰符，成员对象，和成员函数的声明与定义的列表。

### 9.1.2 前置声明

前置声明的语法，如下：

|  |
| --- |
| class-key attr identifier ; |

在作用域内声明一个稍后定义的class类型。直到定义出现，该类都具有不完整的类型。这样做的目的就是，类可以互相引用：

比如，下面的例子：

|  |
| --- |
| class Vector; // 前置声明  class Matrix {  // ...  friend Vector operator\*(const Matrix&, const Vector&);  };  class Vector {  // ...  friend Vector operator\*(const Matrix&, const Vector&);  }; |

在上面的例子中，Matrix的定义里面，声明友邻函数的时候，使用了类型Vector。但是，此时，Vector还没有定义，使用前置声明以后，就可以在编译时先引用类型Vector，等到后面碰见其定义后，再将其完全替代。

如果一个特殊的源文件只使用指针和引用去引用class对象，那么这样做的话，会减少#include的依赖关系，比如，下面的代码：

|  |
| --- |
| // 这是MyStruct.h头文件  #include <iosfwd> // 包含对std::ostream的前置声明  struct MyStruct {  int value;  friend std::ostream& operator<<(std::ostream& os, const S& s);  // 提供定义的源文件MyStruct.cpp里使用#include <ostream>包含std::ostream  }; |

如果前置声明出现在局部范围，它会隐藏先前声明的class，变量，函数，和其它的声明。参考下面的代码：

|  |
| --- |
| struct s { int a; };  struct s; // 没有任何效果(s 已经出现在本作用范围内了)  void g() {  struct s; // 新的前置声明，局部struct "s"，这会隐藏上面全局的结构s，直到该作用块结束。  s\* p; // 指向的是局部结构体s  struct s { char\* p; }; // 局部结构体s的定义  } |

### 9.1.3 成员声明

1.成员声明的语法如下：

|  |
| --- |
| attr(可选) decl-specifier-seq(可选) member-declarator-list(可选) ; |

* attr：属性列表
* decl-specifier-seq：修饰符序列
* member-declarator-list：成员声明

可以声明static、非static数据成员和成员函数，类型别名，枚举，和嵌套类。还可以是友邻声明。如下面的代码所示：

|  |
| --- |
| class S {  int d1; // 非静态数据成员  int a[10] = {1,2}; // 带有初始化的非静态数据成员(C++11)  static const int d2 = 1; // 带有初始化的静态数据成员  virtual void f1(int) = 0; // 纯虚函数  std::string d3, \*d4, f2(int); // 两个数据成员和一个成员函数  enum {NORTH, SOUTH, EAST, WEST};// 枚举  struct NestedS { /\* 嵌套类 \*/  std::string s;  } d5, \*d6;  typedef NestedS value\_type, \*pointer\_type; /\* 类型别名 \*/  }; |

2.函数定义，它们声明和定义成员函数和友邻函数。成员函数定义后面的“；”号是可选的。所有在类体内定义的函数都是内联的（inline）。

|  |
| --- |
| class M {  std::size\_t C;  std::vector<int> data;  public:  M(std::size\_t R, std::size\_t C) : C(C), data(R\*C) {} // 构造函数定义  int operator()(size\_t r, size\_t c) const { // 成员函数定义  return data[r\*C+c];  }  int& operator()(size\_t r, size\_t c) { // 声明和定义一起的成员函数，其是内联的（inline）  return data[r\*C+c];  }  }; |

3.访问修饰符，public:，protected:，和private:

4.Using声明，参考下面的代码：

|  |
| --- |
| class Base {  protected:  int d;  };  class Derived : public Base {  public:  using Base::d; // 使Base的protected成员d成为派生类的一个公共成员  using Base::Base; // 继承父类的所有构造函数(C++11)  }; |

5.static\_assert声明

|  |
| --- |
| template<typename T>  struct Foo {  static\_assert(std::is\_floating\_point<T>::value, "Foo<T>: T must be floating point");  }; |

6.模板成员声明

|  |
| --- |
| struct S {  template<typename T>  void f(T&& n);    template<class CharT>  struct NestedS {  std::basic\_string<CharT> s;  };  }; |

7．别名声明

|  |
| --- |
| template <typename T>  struct identity  {  using type = T;  }; |

### 9.1.4 共同体声明

一个union声明的共同体是一个特殊的class类型，一次只能拥有一个非静态数据成员。

共同体的声明，与class或struct声明类似，如下所示：

|  |
| --- |
| union attr class-head-name { member-specification } |

1. attr：任意数量属性的可选序列。
2. class-head-name：正在定义的共同体的名称。
3. member-specification：访问修饰符、成员对象、和成员函数声明及定义的列表。

共同体可以拥有成员函数（包括构造函数和析构函数），但是不能是虚函数（virtual）。

共同体不能拥有基类，也不能作为基类。

共同体不能拥有引用类型的非静态数据成员。（无法确定其大小。）

和struct结构体一样，默认成员访问是public。

## 9.2 注入类名

注入类名是该类作用域内的类名称。

在类模板中，可以将注入类名用作引用当前模板的模板名称，也可以用作引用当前实例的类名称。

### 9.2.1 解释

在类作用域中，将当前类的名称视为公共成员名称；这被称为注入类名称。该名称声明地方立即跟随着类定义的大括号。

|  |
| --- |
| int X;  struct X {  void f() {  X\* p; // OK。X引用注入类名  ::X\* q; // 错误：名称查找发现一个变量名称，它隐藏了结构体名称  }  }; |

像其他成员一样，注入类名是继承的。 在私有或受保护继承的存在下，间接基类的注入类名可能最终在派生类中不可访问。

|  |
| --- |
| struct A {};  struct B : private A {};  struct C : public B {  A\* p; // Error:注入类名 A 是不可访问的  ::A\* q; // OK, 没有使用注入类名  }; |

### 9.2.2 类模板

像其他类一样，类模板具有注入类名。 注入类名可以用作模板名称或类型名称。

在以下情况下，注入类名被视为类模板本身的模板名称：

* 它后面跟着<
* 它用作模板的模板参数对应的模板参数
* 它是友邻类模板声明的详细类说明符中的最终标识符。

否则，它将被视为类型名称，并等同于模板名称，后跟<>中包含的类模板的模板参数。

|  |
| --- |
| template <template <class, class> class> struct A;    template<class T1, class T2>  struct X {  X<T1, T2>\* p; // OK, X 被看作模板名称  using a = A<X>; // OK, X被看作模板名称  template<class U1, class U2>  friend class X; // OK, X 被看作模板名称  X\* q; // OK, X 被看作类型名，等价于X<T1, T2>  }; |

Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a type-name, it is equivalent to the template-name followed by the template-arguments of the class template specialization or partial specialization enclosed in <>.

在类模板专业化或部分专业化的范围内，当inject-class-name用作类型名称时，它等同于模板名称，后跟类模板特化或部分特化的模板参数 在<>中。

|  |
| --- |
| template<>  struct X<void, void> {  X\* p; // OK, X is treated as a type-name, equivalent to X<void, void>  template<class, class>  friend class X; // OK, X is treated as a template-name (same as in primary template)  X<void, void>\* q; // OK, X is treated as a template-name  };  template<class T>  struct X<char, T> {  X\* p, q; // OK, X is treated as a type-name, equivalent to X<char, T>  using r = X<int, int>; // OK, can be used to name another specialization  }; |

The injected-class-name of a class template or class template specialization can be used either as a template-name or a type-name wherever it is in scope.

类模板或类模板特化的注入类名既可以用作模板名，也可以用作类型名，无论它在何处。

|  |
| --- |
| template<> class X<int, char> {  class B {  X a; // meaning X<int, char>  template<class,class> friend class X; // meaning ::X  };  };  template <class T> struct Base {  Base\* p;  };  template <class T> struct Derived: public Base<T> {  typename Derived::Base\* p; // meaning Derived::Base<T>  };  template<class T, template<class> class U = T::template Base> struct Third { };  Third<Derived<int>> t; // OK: default argument uses injected-class-name as a template |

A lookup that finds an injected-class-name can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations of the same class template, and if the name is used as a template-name, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous.

发现注入类名称的查找可能会在某些情况下导致模糊（例如，如果它在多个基类中找到）。 如果找到的所有注入类名都引用同一类模板的专业化名称，并且该名称被用作模板名称，则引用是指类模板本身，而不是其专门化，而不是 暧昧。

|  |
| --- |
| template <class T> struct Base {};  template <class T> struct Derived: Base<int>, Base<char> {  typename Derived::Base b; // error: ambiguous  typename Derived::Base<double> d; // OK  }; |

### 9.2.3 注入类名和构造函数

Constructors do not have names, but the injected-class-name of the enclosing class is considered to name a constructor in constructor declarations and definitions.

In a qualified name C::D, if

* name lookup does not ignore function names, and
* lookup of D in the scope of the class C finds its injected-class-name

the qualified name is always considered to name C's constructor. Such a name can only be used in the declaration of a constructor (e.g. in a friend constructor declaration, a constructor template specialization, constructor template instantiation, or constructor definition) or be used to inherit constructors (since C++11).

|  |
| --- |
| struct A {  A();  A(int);  template<class T> A(T) {}  };  using A\_alias = A;    A::A() {}  A\_alias::A(int) {}  template A::A(double);    struct B : A {  using A\_alias::A;  };    A::A a; // Error: A::A is considered to name a constructor, not a type  struct A::A a2; // OK, same as 'A a2;'  B::A b; // OK, same as 'A b;' |

## 9.3 类成员

### 9.3.1 非静态数据成员

#### 9.3.1.1 布局

#### 9.3.1.2 标准布局

#### 9.3.1.3 成员初始化

#### 9.3.1.4 使用

#### 9.3.2 非静态函数成员

## 9.4 静态成员- 嵌套类（已完成）

在类定义中，关键字static声明的成员，不会被绑定到该类的实例上。如果位于类定义之外，具有不同的意义。关于其具体意义，可以参考[存储周期](#_5.7_存储周期和链接)。

语法比较简单：

|  |  |
| --- | --- |
| static data\_member | (1) |
| static member\_function | (2) |

解释：

类的静态成员和类的对象没有关系：它们是独立的变量，具有static或thread（C++11）存储周期。

static关键字可以用来声明类成员，但是不可以用来定义类成员，也就是说，类的static成员不能在类内被初始化。比如下面所示：

|  |
| --- |
| class X { static int n; }; // 使用static声明  int X::n = 1; // 不能使用static声明 |

引用类的静态成员，可以使用两种方式，限定访问符“::”或成员访问表达式“->”或“.”。

|  |
| --- |
| struct Foo;  struct S  {  static int a[]; // 声明，不完整类型（因为现在在结构体内，还没有生成对象，所以不能定义）  static Foo x; // 声明，不完整类型  static S s; // 声明，不完整类型（在它自己的定义中）  };    int S::a[10]; // 声明，完整类型  struct Foo {};  Foo S::x; // 声明，完整类型  S S::s; // 声明，完整类型 |

但是，从C++11之后，如果该声明使用了constexpr或inline（C++17）修饰符，成员必须声明为完整类型。

静态成员都遵循[类成员访问规则（private、protected、public）](#_9.7_访问权限)

### 9.4.1 静态成员函数

静态成员函数和任何对象没有关系。当被调用的时候，没有this指针。静态成员函数不能是virtual，const，或volatile。

静态成员函数的地址可以存储在指向函数的常规指针中，但不存储在指向成员函数的指针中。

### 9.4.2 静态数据成员

静态数据成员也和任何对象没有关系。即使没有任何类对象被定义，它们也已经被定义。在整个程序中，静态数据成员只有一个实例，具有静态存储周期，除非使用了关键字thread\_local，在线程存储周期内，每个线程只有一个这样的对象。

静态数据成员不能是mutable。

类中的静态数据成员具有外部链接属性，如果类本身具有外部链接属性的话（例如，不是非命名空间的一个成员）。

## 9.5 派生类 - using-declaration

## 9.6 虚函数和抽象类

## 9.7 访问权限

使用private、protected、和public，定义struct、class或union的成员访问权限。如果是在派生类声明中修饰要继承的基类，定义了继承的基类成员的访问权限。

语法非常简单，就是在类成员声明前加上这3个关键字即可。

|  |  |
| --- | --- |
| public : member-declaration | (1) |
| protected : member-declaration | (2) |
| private : member-declaration | (3) |
| : public base\_class | (4) |
| : protected base\_class | (5) |
| : private base\_class | (6) |

说明：

* + - 1. 成员具有public访问权限，即类、派生类、类对象、派生类对象以及它们的成员函数都可以访问该成员。
      2. 成员具有protected访问权限，即类、派生类的成员函数可以访问，其它的都不可以，包括类或派生类的对象的成员函数都不可以。
      3. 成员具有private访问权限，即只有类的成员函数可以访问，其它的都不可以。包括类、派生类、类对象、派生类对象、派生类成员函数，类对象成员函数，派生类对象成员函数等。
      4. Public继承：基类的public和protected成员对派生类来说，是public和protected成员。
      5. Protected继承：基类的public和protected成员对派生类来说，是protected成员。
      6. Private继承：基类的public和protected成员都成为派生类的private成员。

## 9.8 友邻

## 9.9 重载

### 9.9.1 override修饰符(C++11之后)

就是指定用一个虚函数覆盖掉另一个虚函数。

1. 语法：

标识符override，必须紧跟在成员函数声明或定义的后面。

|  |  |
| --- | --- |
| declarator virt-specifier-seq(可选) pure-specifier(可选) | （1） |
| declarator virt-specifier-seq(可选) function-body(可选) | （2） |

1）在成员函数声明中，override位于virt-specifier-seq中，在pure-specifier之前。

2）在成员函数定义中，override位于virt-specifier-seq中，在function-body之前。

在这两种情况中，virt-specifier-seq既可以是override或final，也可以是它们的组合。

2. 解释

在成员函数声明或定义中，override确保该函数是虚函数并从基类重写虚函数。如果不是，程序格式就会发生错误（编译时错误）。

当在成员函数的声明符后面，使用override时，具有特殊的意义：要不然，它就不是一个保留关键字了。

3. 示例

|  |
| --- |
| struct A  {  virtual void foo();  void bar();  };  struct B : A  {  void foo() const override; // 错误：B::foo 不能重载A::foo（符号不匹配）  void foo() override; // OK: B::foo重载A::foo  void bar() override; // 错误: A::bar不是虚函数  }; |

### 9.9.2 final修饰符(C++11之后)

指定无法在派生类中重载虚函数，或者类无法被继承。

1 语法

应用到成员函数身上时，标识符final紧跟在成员函数声明或成员函数定义语法中的声明符的后面。

应用到Class上时，标识符final紧跟在类名称的后面，类定义的开始处。

|  |  |
| --- | --- |
| declarator virt-specifier-seq(可选) pure-specifier(可选) | （1） |
| declarator virt-specifier-seq(可选) function-body | （2） |
| class-key attr(可选) class-head-name class-virt-specifier(可选) base-clause(可选) | （3） |

（1）在成员函数声明中，final位于virt-specifier-seq中

（2）在成员函数的定义中，final位于virt-specifier-seq中

（3）在类定义中，final出现在类名称后面的class-virt-specifier中，且在基类子句后面

在情况（1）和（2）中，virt-specifier-seq可以是override或final，也可以是它们的组合。在情况（3）中，只能是final。

2. 解释

当在虚函数声明或定义中使用时，final确保函数是虚函数并指定它不会被派生类覆盖。否则，发生语法错误（编译时报错）。

当在类定义中使用时，final指定此类可能不会出现在另一个类定义的base-specifier-list中（换句话说，不能从中派生）。否则，发生语法错误（编译时报错）。final也可以与union定义一起使用，在这种情况下它没有效果（除了std::is\_final的结果），因为unions不能被派生）

final是在成员函数声明或类头中使用时具有特殊含义的标识符。 在其它上下文中，它不是保留关键字，可用于命名对象和函数。

|  |
| --- |
| struct Base  {  virtual void foo();  };    struct A : Base  {  void foo() final; // A::foo 被重载，不能再被重载  void bar() final; // 错误：非虚函数不能被重载或禁止重载  };    struct B final : A // struct B is final  {  void foo() override; // Error: foo cannot be overridden as it's final in A  };    struct C : B // Error: B is final  {  }; |

## 9.10 位域-this指针

### 9.10.1 位域

按位声明一个具有显式大小的类数据成员。相邻的位域成员可以打包共享和跨越单个字节。

位域声明是使用以下声明符的类数据成员声明：

|  |  |  |
| --- | --- | --- |
| 标识符(可选) 属性字段(可选):大小 | （1） |  |
| 标识符(可选) 属性字段(可选):大小 括号或等号初始化 | （2） | C++20之后 |

位域的类型由声明语法的decl-specifier-seq引入。

* 属性字段： 可选的任意数量的属性序列
* 标识符：正在声明的位域的名称。该名称是可选的：无名位域引入了指定的填充位数。
* 大小： 一个大于或等于零的整数常量表达式。 当大于零时，这是该位字段将占用的位数。值零只能用于无名的位域，具有特殊含义：它指定类定义中的下一个位域将从分配单元的边界开始。
* 括号或等号初始化： 使用[默认成员初始化](#_9.3.1.3_成员初始化)。

1.解释

（1）位域中的位数限制值的容纳范围：

|  |
| --- |
| #include <iostream>  struct S {  // 3位无符号位域，允许的值为0...7  unsigned int b : 3;  };  int main()  {  S s = {7};  ++s.b; // 无符号数溢出  std::cout << s.b << '\n'; // 输出结果为：0  } |

（2）通常将多个相邻位域打包在一起（尽管此行为是执行时定义的）：

|  |
| --- |
| #include <iostream>  struct S {  // 通常会占据2个字节  // 3 位： b1  // 2 位： 未使用  // 6 位： b2  // 2 位： b3  // 3 位： 未使用  unsigned char b1 : 3, : 2, b2 : 6, b3 : 2;  };  int main()  {  std::cout << sizeof(S) << '\n'; // 通常会打印2  } |

（3）大小为零的特殊未命名位域可以被强制分割填充。它指定下一个位域从其分配单元的开始处开始：

|  |
| --- |
| #include <iostream>  struct S {  unsigned char b1 : 3;  unsigned char :0; // 启动一个新的字节  unsigned char b2 : 3;  unsigned char b3 : 1;  };  int main()  {  std::cout << sizeof(S) << '\n'; // 通常会打印2  } |

* 如果位域的指定大小大于其类型的大小，则该值受其类型的限制：例，std::uint8\_t b：1000;仍然保持0到255之间的值。额外的位变成未使用的填充。
* 由于位域不一定始于字节的开头，所以不能采用位域的地址。指向位域的指针和非const引用是不可以的。当使用位域[初始化const引用](#_6.7_引用初始化)时，会创建一个临时对象（它的类型是位域的类型），用位域的值完成拷贝赋值，引用绑定到该临时对象。
* 位域的类型只能是整形或者枚举类型。
* 位域不能是[静态数据成员](#_9.4_Static_members)。
* 没有位域类型纯右值：左值到右值的转换总是生成位域的基础类型的对象。

|  |  |
| --- | --- |
| 位域没有默认成员初始化：int b : 1 = 0;和int b : 1{0}都是不规范的。 | C++20之前 |
| 如果位域的大小和默认成员初始化之间存在歧义，那么选择构成有效大小的最长符号序列：   |  | | --- | | int a;  const int b = 0;  struct S {  // 简单的情况  int x1 : 8 = 42; // OK; "= 42"赋值初始化  int x2 : 8 { 42 }; // OK; "{ 42 }" 括号初始化  // 有歧义  int y1 : true ? 8 : a = 42; // OK; 初始化不存在  int y2 : true ? 8 : b = 42; // 错误:不能赋值const int  int y3 : (true ? 8 : b) = 42; // OK; "= 42"赋值初始化  int z : 1 || new int { 0 }; // OK; 不存在初始化  }; | | C++20之后 |

2 注意

位域的下列属性是执行时定义的：

* 用超出范围的值，赋值或初始化有符号位域，或通过对有符号位域进行加法运算超过其范围而产生的值。
* 关于类对象内位域的实际分配细节的一切。
  + 例如，在某些平台上，位域不会跨越字节，而在其它的平台上就可以；
  + 另外，在某些平台上，位域从左到右打包，其它的平台上从右到左打包；
* 未明确说明是有符号还是无符号的char，short，int，long和long long位域是有符号还是无符号（C++14之前）
  + 例如，int b：3; 可能具有值0..7或-4..3的范围。

在C编程语言中，位域的宽度不能超过其类型的宽度。

### 9.10.2 this指针

#### 9.10.2.1 语法

关键字this是一个纯右值表达式，其值是对象的地址，在该对象上调用成员函数。它能出现在下列语境中：

1. 任何非静态成员函数体内，包含成员初始化列表
2. 在非静态成员函数的声明内任何(可选)cv限定符序列后处，包含动态异常规范(弃用)、 noexcept规范(C++11)及拖尾返回类型(C++11之后)
3. 造[默认成员初始化](#_6.1_默认初始化)中(C++11之后)

类X的成员函数中this的类型是X\*（指向X的指针）。如果成员函数是cv限定的，this的类型是cv X\*。因为构造函数与析构函数不能为cv限定，故它们之中this的类型始终是X\*，即使在构造或销毁const对象时。

当非静态数据成员被用于任何允许this关键词的语境中时（非静态成员函数体、成员初始化器列表、默认成员初始化器），名称前会自动隐式地添加this->，产生一个成员访问表达式（这里若成员是虚函数，则产生虚函数调用）。

In class templates, this is a dependent expression, and explicit this-> may be used to force another expression to become dependent.

在类模板中，这是一个依赖表达式，而且显式的 this-> 可以用于强制另一表达式变为依赖的。

在一个对象的构造期间，若通过不直接或间接从构造函数的 this 指针获得的泛左值，访问对象的值或任何其子对象，则如此获得的对象或子对象的值是未指定的。换言之，构造函数中 this 指针不能别名使用：

During construction of an object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor's this pointer, the value of the object or subobject thus obtained is unspecified. In other words, the this pointer cannot be aliased in a constructor:

|  |
| --- |
| extern struct D d;  struct D {  D(int a) : a(a), b(d.a) {} // b(a)或b(this->a)是正确的  int a, b;  };  D d = D(1); // 因为b(d.a)不通过this访问， d.b现在是未指定的 |

It is possible to execute delete this;, if the program can guarantee that the object was allocated by new, however, this renders every pointer to the deallocated object invalid, including the this pointer itself: after delete this; returns, such member function cannot refer to a member of a class (since this involves an implicit dereference of this) and no other member function may be called. This is used, for example, in the member function of the control block of std::shared\_ptr responsible for decrementing the reference count, when the last reference to the managed object goes out of scope.

|  |
| --- |
| class ref  {  // ...  void incRef() { ++mnRef; }  void decRef() { if (--mnRef == 0) delete this; }  }; |

#### 9.10.2.2 示例

|  |
| --- |
| class T  {  int x;    void foo()  {  x = 6; // 与this->x = 6;相同  this->x = 5; // 显式使用this->  }    void foo() const  {  // x = 7; // Error: \*this is constant  }    void foo(int x) // 参数x屏蔽具有相同名称的成员  {  this->x = x; // 非限定的x引用的是传递进来的参数，要求用this->消除歧义  }    int y;  T(int x) : x(x), // 使用参数x初始化成员x  y(this->x) // 使用成员x初始化变量y  {}    T& operator= ( const T& b )  {  x = b.x;  return \*this; // 许多重载运算符返回\*this  }  };    class Outer {  int a[sizeof(\*this)]; // 错误: 不是在成员函数里  unsigned int sz = sizeof(\*this); // OK:在默认成员初始化器中  void f() {  int b[sizeof(\*this)]; // OK  struct Inner {  int c[sizeof(\*this)]; // 错误: 没有在Inner的成员函数中  };  }  } |

## 9.11 构造函数和成员初始化列表

## 9.12 默认构造函数和析构函数

## 9.13 拷贝构造函数 - Copy assignment

## 9.14 移动构造函数(C++11) - Move assignment(C++11)

如果一个构造函数，没有使用说明符explicit进行修饰，被叫做转换构造函数（converting constructor）。

## 9.15 转换构造函数（已完成）

### 9.15.1 解释与说明

构造函数如果没有使用关键字explicit进行修饰，则被称为转换构造函数。在C++11之前，只有具有一个参数的构造函数才称为转换构造函数。

显式构造函数只有直接初始化（包括显式转换，比如static\_cast）才能调用，而转换构造函数还可以在拷贝初始化的时候使用，成为用户自定义转换的一种。

### 9.15.2 示例

|  |
| --- |
| struct A  {  A() {} // 转换构造函数（C++11之后）  A(int) {} // 转换构造函数  A(int, int) {} // 转换构造函数（C++11之后）  };  struct B  {  explicit B() {}  explicit B(int) {}  explicit B(int, int) {}  };  int main(void)  {    A a\_1 = 1; // OK:拷贝初始化会选择A::A(int)  A a\_2(2); // OK:直接初始化会选择A::A(int)  A a\_3{4,5}; // OK:直接列表初始化会选择A::A(int,int)  A a4 = {4, 5}; // OK:拷贝列表初始化会选择A::A(int, int)  A a5 = (A)1; // OK:显式强制转换，相当于static\_cast，直接初始化  // B b1 = 1; // error:拷贝初始化不考虑B::B(int)  B b2(2); // OK:直接初始化会选择B::B(int)  B b3{4, 5}; // OK:直接列表初始化会选择B::B(int, int)  // B b4 = {4, 5}; // error:拷贝列表初始化会选择一个显式constructor  // B::B(int, int)  B b5 = (B)1; // OK:显式强制转换，相当于static\_cast，直接初始化  B b6; // OK,默认初始化  B b7{}; // OK,直接列表初始化  // B b8 = {}; // error:拷贝列表初始化会选择一个显式constructor  // B::B()  return 0;  } |

### 9.15.3 关键字explicit

# 10 模板

模板是C++实体的一种，可以是如下定义之一：

* 类模板，可以是嵌套类
* 函数模板，可以是成员函数
* 别名模板(C++11添加)
* 变量模板(C++14添加)
* [a concept (constraints and concepts) (since C++20)](#_10.13_约束和概念(C++20)(修改中))

模板有3种参数：类型模板参数，非类型模板参数，模板模板参数（以模板作为模板的参数）。

模板的特化过程就是，向模板提供实参，替换掉模板的形参，进而将模板特化的过程。换句话说，就是将模板变为一个特定的类型或函数左值。还可以显式特化：类和函数模板都允许完全特化，部分特化只有类模板可以。

当在需要完整对象类型的上下文中引用类模板特化时，或者在需要存在函数定义的上下文中引用函数模板特化时，将实例化模板（实际上，就是编译它的代码），除非模板已经显式特化或显式实例化。 除非类的成员函数也被使用，否则类模板的实例化不会实例化其任何成员函数。在链接时，合并由不同翻译单元生成的相同实例。

模板的定义必须在隐式实例化时可见，这就是模板库通常在头文件中提供所有模板定义的原因（例如，大多数boost库只是头文件）

1.语法

|  |  |  |
| --- | --- | --- |
| template < parameter-list > requires-clause(C++20)(可选) declaration | （1） |  |
| export template < parameter-list > declaration | （2） | C++11之前 |
| template < parameter-list > concept concept-name = constraint-expression ; | （3） | C++20之后 |

* declaration：模板声明
* parameter-list：参数列表
* concept-name：查看constraints and concepts(since C++20)

2.模板标识符（template-id）

template-name：模板名称，既可以是命名模板的标识符也可以是重载运算符模板或用户定义的文本模板的名称。

|  |
| --- |
| template<class T, T::type n = 0> class X;  struct S {  using type = int;  };  using T1 = X<S, int, int>; // 错误: 太多实参  using T2 = X<>; // 错误: 第一个模板形参没有默认参数  using T3 = X<1>; // 错误: 值1 不能匹配type形参  using T4 = X<int>; // 错误: 第二个模板形参替换失败  using T5 = X<S>; // OK |

## 10.1 模板形参和实参

### 10.1.1 模板形参（已完成\*\*）

#### 10.1.1.1 无类型模板形参

|  |  |  |
| --- | --- | --- |
| type name(可选) | （1） |  |
| type name(可选) = default | （2） |  |
| type ... name(可选) | （3） | C++11之后 |
| **auto** name | （4） | C++17之后 |

1. 无类型模板形参，name可以省略。
2. 具有可选名称和默认值的无类型模板参数。
3. 具有可选名称的无类型模板参数包。
4. 具有推导类型的无类型模板参数（注意，它也可以是auto \*\*或包含占位符auto的任何其他类型）

无类型模板形参type可以是以下类型中的任何一个（可以选择cv-qualified，忽略限定符）：

1. std::nullptr\_t（C++11之后）
2. 整数类型；
3. 左值引用类型（引用对象或者函数）；
4. 指针类型（指向对象或函数）；
5. 成员指针类型（指向成员对象或成员函数）；
6. 枚举类型。

数组和函数类型可以写入模板声明中，但它们会自动替换为指向对象和指针的指针。

在类模板正文中的表达式中使用非类型模板参数的名称时，除非其类型是左值引用类型，否则它是不可修改的纯右值。

|  |  |
| --- | --- |
| 如果包含占位符关键字auto，则可以推导出非类型模板参数的类型： | C++17之后 |
| template<**auto** n> struct B { /\* ... \*/ };  B<5> b1; // 可以：非类型模板参数是int  B<'a'> b2; // 可以：非类型模板参数是char  B<2.5> b3; // 错误：非类型模板参数不可以是double |

**核心笔记：无类型模板参数不能是对象，甚至不能是double或者float。无类型参数仅限于int、enmu、指针和引用。有时，可能想让用户指定一个特定类型的值来初始化空对象，就可以使用无类型模板参数：**

|  |
| --- |
| template<typename T,const T EMPTY>  class Grid  {  public:  // ...  Grid(const Grid<T,EMPTY>& src);  Grid<T,EMPTY>& operator=( const Grid<T,EMPTY>& rhs);  // ...  }; |

对于上面的代码，我们可以这样使用

|  |
| --- |
| Grid<int,10> myIntGrid;  Grid<int,20> myIntGrid2; |

#### 10.1.1.2 类型模板参数

|  |  |  |
| --- | --- | --- |
| **typename|class** name(可选) | （1） |  |
| **typename|class** name(可选) = default | （2） |  |
| **typename|class** ... name(可选) | （3） | C++11之后 |

typename和class在类型模板参数中没有什么差别。

1. 没有默认值的类型模板参数

|  |
| --- |
| template<class T>  class My\_vector{ /\* … \*/ } |

1. 带有默认值的类型模板参数

|  |
| --- |
| template<class T = void>  struct My\_op\_functor { /\* ... \*/ }; |

1. 类型模板参数包。

|  |
| --- |
| template<typename... Ts>  class My\_tuple { /\* ... \*/ }; |

在模板声明的主体中，类型参数的名称是一个typedef-name，用于替换模板实例化时提供的类型。

#### 10.1.1.3 模板类型的模板参数

语法：

|  |  |  |
| --- | --- | --- |
| **template**<parameter-list> **typename**(C++17)|**class** name(可选) | （1） |  |
| **template**<parameter-list> **typename**(C++17)|**class** name(可选)=default | （2） |  |
| **template**<parameter-list> **typename**(C++17)|**class** ... name(可选) | （3） | C++11之后 |

1. 具有可选名称的模板类型的模板参数。
2. 带有可选名称和默认值的模板类型的模板参数。
3. 具有可选名称的模板类型的模板参数包。

|  |  |
| --- | --- |
| 与类型模板参数声明不同，模板的模板参数声明只能使用关键字**class**而不使用**typename**。 | C++17之前 |

在模板声明的主体中，此参数的名称是一个模板名称（并且需要实例化参数）。

|  |
| --- |
| template<typename T> class my\_array {};    // 两个类型模板参数和一个模板类型的模板参数：  template<typename K, typename V, template<typename> typename C = my\_array>  class Map  {  C<K> key;  C<V> value;  }; |

#### 10.1.1.4 约束模板参数（C++20之后）

### 10.1.2 模板实参

为了使模板实例化，每个模板参数（类型，无类型或模板）都必须由相应的模板实参替换。对于类模板，参数要么显式提供，要么从初始化中推导（从C++17开始），或使用默认值。对于函数模板，参数要么显式提供，要么从上下文中推导，或使用默认值。

如果一个模板参数是类型表达式，即包括type-id和表达式，即使相应的模板参数是非类型的（下面的代码中的#2），它也总是被解释为一个类型模板参数：

|  |
| --- |
| template<class T> void f(); // #1  template<int I> void f(); // #2  void g() {  f<int()>(); // "int()" 即是类型也是表达式  // 调用#1因为它被解释为类型  } |

#### 10.1.2.1 无类型实参

实例化具有非类型模板参数的模板时，以下限制适用：

|  |  |
| --- | --- |
| * 对于整型和算术类型，在实例化过程中，提供的模板实参必须是模板参数类型的转换的常量表达式（因此，会应用某些隐式转换）。 * 对于指向对象的指针，模板实参必须指定为具有static存储周期和链接（内部或外部）的完整对象的地址，或指定为适当的空指针或std :: nullptr\_t值的常量表达式。 * 对于指向函数的指针，有效实参是指向具有链接属性的函数的指针（或者计算为空指针值的常量表达式）。 * 对于左值引用参数，在实例化时提供的实参不能是临时变量，或未命名的左值或没有链接属性的命名左值（换句话说，参数必须具有链接属性）。 * 对于成员指针，实参必须是一个指向成员的指针，该指针表示为＆Class::Member或者一个常量表达式，其值为空指针或std :: nullptr\_t值   特别是，这意味着字符串文字，数组元素的地址和非静态成员的地址不能用作模板实参来实例化其相应的非类型模板参数是指向对象的指针的模板。 | C++17之前 |
| 可以与非类型模板参数一起使用的模板实参可以是任何已转换的模板参数类型的常量表达式。   |  | | --- | | template<const int\* pci> struct X {};  int ai[10];  X<ai> xi; // ok: 数组向指针转换和cv限定的转换    struct Y {};  template<const Y& b> struct Z {};  Y y;  Z<y> z; // ok: 没有转换    template<int (&pa)[5]> struct W {};  int b[5];  W<b> w; // ok: 没有转换    void f(char);  void f(int);  template<void (\*pf)(int)> struct A {};  A<&f> a; // ok: 重载决策选择f(int) |   唯一的例外是引用和指针类型的非类型模板参数不能引用/指向   * 子对象（包括非静态类成员，基类子对象或数组元素）; * 临时对象（包括在引用初始化期间创建的对象）; * 一个字符串文字; * typeid的结果; * 或预定义变量\_\_func\_\_。  |  | | --- | | template<class T, const char\* p> class X {};  X<int, "Studebaker"> x1; // 错误: 字符串不能作为模板实参    template<int\* p> class X {};  int a[10];  struct S  {  int m;  static int s;  } s;  X<&a[2]> x3; // 错误: 数组元素的地址  X<&s.m> x4; // 错误: 非静态成员地址  X<&s.s> x5; // ok: 静态成员地址  X<&S::s> x6; // ok: 静态成员地址    template<const int& CRI> struct B {};  B<1> b2; // 错误: 临时值作为模板实参  int c = 1;  B<c> b1; // ok | | C++17之后 |

#### 10.1.2.2 类型实参

类型模板参数的模板参数必须是一个type-id，它可以命名为一个不完整的类型：

|  |
| --- |
| template<typename T> class X {}; // 类型模板    struct A; // 不完整类型  typedef struct {} B; // 未命名的类型别名    int main()  {  X<A> x1; // ok: 'A' 命名了一个类型  X<A\*> x2; // ok: 'A\*' 命名了一个类型  X<B> x3; // ok: 'B' 命名了一个类型  } |

#### 10.1.2.3 模板作为实参

A template argument for a template template parameter must be an id-expression which names a class template or a template alias.

When the argument is a class template, only the primary template is considered when matching the parameter. The partial specializations, if any, are only considered when a specialization based on this template template parameter happens to be instantiated.

模板模板参数的模板参数必须是用于命名类模板或模板别名的id表达式。

当实参是class模板时，匹配参数时仅考虑主模板。部分特化，如果有的话，仅在基于此模板模板参数的专业化碰巧实例化时才被考虑。

|  |
| --- |
| template<typename T> class A { int x; }; // primary template  template<class T> class A<T\*> { long x; }; // 部分特化    // class template with a template template parameter V  template<template<typename> class V> class C  {  V<int> y; // uses the primary template  V<int\*> z; // uses the partial specialization  };    C<A> c; // c.y.x has type int, c.z.x has type long |

（未完？？？？）

#### 10.1.2.4 默认实参

默认模板参数在=符号后面的参数列表中指定。 可以为任何类型的模板参数（类型，非类型或模板）指定默认值，但不能指定参数包。

（未完？？？？）

### 10.1.3 举例

非类型模板参数：

|  |
| --- |
| #include <iostream>  // 简单的非类型模板参数  template<int N>  struct S { int a[N]; };  template<const char\*>  struct S2 {};  // 复杂的非类型模板参数  template  <  char c, // 整数类型  int (&ra)[5], // 左值引用对象(数组类型)  int (\*pf)(int), // 函数指针  int (S<10>::\*a)[10] // 成员指针 (类型int[10])  > struct Complicated  {  // 调用编译时选择的函数 且存储结果到编译时选中的数组  void foo(char base)  {  ra[4] = pf(c - base);  }  };    // S2<"fail"> s2; // 错误: 字符串文本不能被使用  char okay[] = "okay"; // 具有链接属性的static对象  //S2< &okay[0] > s2; // 错误: 数组元素没有链接属性  S2<okay> s2; // 可以工作    int a[5];  int f(int n) { return n; }    int main()  {  S<10> s; // s.a是10个int元素的数组  s.a[9] = 4;    Complicated<'2', a, f, &S<10>::a> c;  c.foo('0');    std::cout << s.a[9] << a[4] << '\n';  } |

输出：

|  |
| --- |
| 42 |

## 10.2 类模板

类模板用来定义一簇类。

### 10.2.1 语法

|  |  |  |
| --- | --- | --- |
| template <参数列表> 类声明 | （1） |  |
| export template <参数列表> 类声明 | （2） | C++11之前 |

解释：

* 类声明 – 如类声明，声明的类名称就是模板的名称。
* 参数列表 – 一个非空的用逗号“，”分割的模板参数列表，可以是非类型参数，类型参数，模板参数或者参数包中的任何一个。

|  |  |
| --- | --- |
| **export**是一个可选的修饰符，它将模板声明为导出（当与类模板一起使用时，它声明所有的成员都已导出）。实例化导出模板的文件不需要包含它们的定义：声明就足够了。export的实施是罕见的，并在细节上没有达成一致。 | C++11之前 |

### 10.2.2 实例化

类模板本身不是类型，对象或任何其他实体。只包含模板定义的源文件编译时是不会生成任何代码的。 为了生成代码，必须实例化模板：必须提供模板实参，以便编译器可以生成实际的类（或函数模板中的函数）。

#### 10.2.2.1 显式实例化

1. 语法

|  |  |  |
| --- | --- | --- |
| **template class**|**struct** 模板名称<实参列表> ; | （1） |  |
| **extern template class**|**struct**模板名称<实参列表> ; | （2） | C++11之后 |

（1）显式实例化定义；

（2）显式实例化声明；

显式实例化定义强制实例化它们引用的类，结构或联合体。它可能出现在模板定义之后的任何地方，对于给定的参数列表，只允许在整个程序中出现一次。

|  |  |
| --- | --- |
| 显式实例化声明（extern模板）会跳过隐式实例化步骤：否则会导致隐式实例化的代码会使用其他位置提供的显式实例化定义替代（如果不存在此类实例化会导致链接错误）。 这可以用来减少编译时间，方法是在除了一个使用它的源文件之外的所有源文件中显式声明一个模板实例化，并在其余文件中显式定义它。 | C++11之后 |

类，函数，变量和成员模板特化可以通过它们的模板显式实例化。类模板的成员函数，成员类和静态数据成员可以从其成员定义中显式实例化。

显式实例化只能出现在模板的封闭名称空间中，除非它使用qualified-id：

|  |
| --- |
| namespace N {  template<class T> class Y { void mf() { } }; // 模板定义  }  // template class Y<int>; // 错误: 类模板Y在全局命名空间中是不可见的  using N::Y;  // template class Y<int>; // 错误: 模板命名空间之外显式实例化  template class N::Y<char\*>; // OK: 显式实例化  template void N::Y<double>::mf(); // OK: 显式实例化 |

Explicit instantiation has no effect if an explicit specialization appeared before for the same set of template arguments.

Only the declaration is required to be visible when explicitly instantiating a function template, a variable template, a member function or static data member of a class template, or a member function template. The complete definition must appear before the explicit instantiation of a class template, a member class of a class template, or a member class template, unless an explicit specialization with the same template arguments appeared before.

If a function template, variable template, member function template, or member function or static data member of a class template is explicitly instantiated with an explicit instantiation definition, the template definition must be present in the same translation unit.

When an explicit instantiation names a class template specialization, it serves as an explicit instantiation of the same kind (declaration or definition) of each of its non-inherited non-template members that has not been previously explicitly specialized in the translation unit. If this explicit instantiation is a definition, it is also an explicit instantiation definition only for the members that have been defined at this point.

Explicit instantiation definitions ignore member access specifiers: parameter types and return types may be private.

如果显式特化出现在同一组模板参数之前，则显式实例化不起作用。

只有当明确实例化函数模板，变量模板，类模板的成员函数或静态数据成员或成员函数模板时，才需要声明。完整的定义必须出现在类模板，类模板的成员类或成员类模板的显式实例化之前，除非之前出现了具有相同模板参数的显式特化。

如果类模板的函数模板，变量模板，成员函数模板或成员函数或静态数据成员通过显式实例化定义显式实例化，则模板定义必须存在于同一个翻译单元中。

当一个显式的实例化命名一个类模板特化时，它作为一个明确的实例化，它的每个非继承的非模板成员的相同类型（声明或定义）在先前没有在转换单元中明确专用。如果这个显式实例化是一个定义，那么它也是一个明确的实例化定义，仅限于此时定义的成员。

显式实例化定义忽略成员访问说明符：参数类型和返回类型可能是私有的。

#### 10.2.2.2 隐式实例化

当代码在需要完全定义类型的上下文中引用模板时，或者当类型的完整性影响代码时，并且此特定类型尚未显式实例化时，会发生隐式实例化。 例如，当这种类型的对象被构造时，而不是当构造这种类型的指针时。

这适用于类模板的成员：除非该成员在程序中使用，否则不会实例化，也不需要定义。

|  |
| --- |
| template<class T> struct Z {  void f() {}  void g(); // 没有定义  }; // 模板定义  template struct Z<double>; // Z<double>显式实例化  Z<int> a; // Z<int>隐式实例化  Z<char>\* p; // 什么都没实例化  p->f(); // Z<char> 和 Z<char>::f()隐式实例化  // Z<char>::g() // 永远不需要，也不需要实例化：它没有定义 |

如果类模板被声明，但是没有定义，在实例化的时候，会产生不完整的类类型：

|  |
| --- |
| template<class T> class X; // 声明，没定义  X<char> ch; // 错误：不完整的类型X<char> |

|  |  |
| --- | --- |
| 局部类和其成员中使用的任何模板都被实例化为声明的局部类或枚举的实体的实例化的一部分。 | C++17之后 |

## 10.3 函数模板

### 10.3.1 语法

|  |  |  |
| --- | --- | --- |
| **template** <参数列表> 函数声明 | （1） |  |
| **template** <参数列表> **requires** constraint function-declaration | （2） | C++20之后 |
| function-declaration-with-placeholders | （3） | concepts TS |
| **export** **template** <参数列表> 函数声明 | （4） | C++11之前 |

解释：

* 参数列表 - 一个非空的用逗号“，”分割的模板参数列表，可以是非类型参数，类型参数，模板参数或者参数包中的任何一个。与任何模板一样，参数可能受到限制（C++20之后）。
* 函数声明 – 声明的函数名称作为模板名称。
* constraint(C++20) - a [constraint expression](#_10.13_约束和概念(C++20)) which restricts the template parameters accepted by this function template
* function-declaration-with-placeholders(concepts TS) - a function declaration where the type of at least one parameter uses the placeholder auto or a constrained type specifier: the template parameter list will have one invented parameter for each placeholder.

|  |  |
| --- | --- |
| **export**是一个可选的修饰符，它将模板声明为导出（当与类模板一起使用时，它声明所有的成员都已导出）。实例化导出模板的文件不需要包含它们的定义：声明就足够了。export的实施是罕见的，并在细节上没有达成一致。 | C++11之前 |

|  |  |
| --- | --- |
| When placeholders (either auto or constrained placeholders) appear in the parameter list of a function declaration or of a function template declaration, one invented template parameter for each placeholder is appended to the template parameter list  当占位符（auto或constrained占位符）出现在函数声明或函数模板声明的参数列表中时，每个占位符的一个发明模板参数会附加到模板参数列表 | concepts TS |

### 10.3.2 实例化

## 10.4 类成员模板

Template declarations (class, function, and variables (since C++14)) can appear inside a member specification of any class, struct, or union that aren't local classes.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>    struct Printer {  std::ostream& os;  Printer(std::ostream& os) : os(os) {}  template<typename T>  void operator()(const T& obj) { os << obj << ' '; } // 成员模板  };    int main()  {  std::vector<int> v = {1,2,3};  std::for\_each(v.begin(), v.end(), Printer(std::cout));  std::string s = "abc";  std::for\_each(s.begin(), s.end(), Printer(std::cout));  } |

输出：

|  |
| --- |
| 1 2 3 a b c |

Partial specializations of member template may appear both at class scope and at enclosing namespace scope, but explicit specializations may only appear at enclosing namespace scope.

|  |
| --- |
| struct A {  template<class T> struct B; // 原始成员模板  template<class T> struct B<T\*> { }; // 正确: 局部特化  // template<> struct B<int\*> { }; // 错误: 完全特化  };  template<> struct A::B<int\*> { }; // 正确  template<class T> struct A::B<T&> { }; // 正确 |

## 10.5 Variable template(C++14)

## 10.6 模板参数推导

## 10.7 显式特化

允许为给定的一组模板参数自定义模板代码。

语法：

|  |
| --- |
| template <> declaration |

能够显式特化的模板：

函数模板

类模板

变量模板（C++14后）

类模板的成员函数

类模板的静态数据成员

类模板的成员类

类模板的枚举成员

类或类模板的成员类模板

类或类模板的成员函数模板

例如，

|  |
| --- |
| #include <iostream>  template<typename T> // 原始模板  struct is\_void : std::false\_type  {  };  template<> // 为T = void显式特化  struct is\_void<void> : std::true\_type  {  };  int main()  {  // for any type T other than void, the  // class is derived from false\_type  std::cout << is\_void<char>::value << '\n';  // but when T is void, the class is derived  // from true\_type  std::cout << is\_void<void>::value << '\n';  } |

### 10.7.1 详细内容

### 10.7.2 函数模板的显式特化

### 10.7.3 模板特化的成员

## 10.8 类模板参数推导(C++17)

## 10.9 局部特化

允许为给定类别的模板参数定制类模板。

### 10.9.1语法：

|  |
| --- |
| template <参数列表> class-key class-head-name <实参列表> 声明 |

where class-head-name identifies the name of a previously declared class template. This declaration must be in the same namespace or, for member templates, class scope as the primary template definition which it specializes .

其中class-head-name标识先前声明的类模板的名称。此声明必须位于相同的名称空间中，或者对于成员模板，类声明必须与其专用的主模板定义相同。

例如，

|  |
| --- |
| template<class T1, class T2, int I>  class A {}; // 主模板  template<class T, int I>  class A<T, T\*, I> {}; // #1: 局部特化，在这里T2是指向T1的指针  template<class T, class T2, int I>  class A<T\*, T2, I> {}; // #2: 局部特化，在这里T1是一个指针  template<class T>  class A<int, T\*, 5> {}; // #3: 局部特化，在这里T1是int, I是5,T2是一个指针  template<class X, class T, int I>  class A<X, T\*, I> {}; // #4: 局部特化，在这里T2是一个指针 |

### 10.9.2 实参列表

### 10.9.3 名称查找

### 10.9.4 局部特化顺序

### 10.9.5 局部特化的成员

## 10.10 Parameter pack

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A function parameter pack is a function parameter that accepts zero or more function arguments.

A template with at least one parameter pack is called a variadic template.

## 10.11 Fold-expressions(C++17)

## 10.12 Dependent names - SFINAE

## 10.13 约束和概念(C++20)(修改中)

|  |
| --- |
| 注意1：正在进行中，此页面正在更新中，以反映C ++ 20工作草案中包含的概念技术规范的各个部分。 请注意，C ++ 20的概念与概念TS在几个重要方面有所不同，目前没有编译器支持它。  注意2：**实验特性** 本页所描述的功能是概念技术规范ISO / IEC TS 19217：2015（概念TS）的一部分， |

# 11 异常

## 11.1 抛出异常（throw）

## 11.2 try-catch

## 11.3 function-try

## 11.4 noexcept specifier(C++11)

## 11.5 noexcept operator(C++11)

## 11.6 Dynamic exception specification(until C++17)

# 12 其它

## 12.1 C++历史

## 12.2 扩展命名空间std

## 12.3 未定义行为

## 12．4 RAII - Rule of three/five/zero

## 12.5 As-if规则

### 12.5.1 As-if规则

允许所有代码的优化，其不会改变程序的可见的行为。

#### 12.5.1.1 解释

只要满足以下条件，C++编译器就可以对程序执行任何更改：

|  |  |
| --- | --- |
| （1）At every sequence point, the values of all volatile objects are stable (previous evaluations are complete, new evaluations not started) | C++11之前 |
| （1）Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread. | C++11之后 |

（2）At program termination, data written to files is exactly as if the program was executed as written.

在程序终止时，写入文件的数据就像程序按照写入的方式执行一样。

（3）Prompting text which is sent to interactive devices will be shown before the program waits for input

提示发送到交互设备的文本将在程序等待输入前显示

（4）If the ISO C pragma #pragma STDC FENV\_ACCESS is supported and is set to ON, the changes to the floating-point environment (floating-point exceptions and rounding modes) are guaranteed to be observed by the floating-point arithmetic operators and function calls as if executed as written, except that

如果支持ISO C pragma #pragma STDC FENV\_ACCESS并将其设置为ON，则通过浮点算术运算符和函数调用可以确保将对浮点环境（浮点异常和舍入模式）的更改视为 如果按照书面执行，除外

the result of any floating-point expression other than cast and assignment may have range and precision of a floating-point type different from the type of the expression (see FLT\_EVAL\_METHOD)

除了强制转换和赋值之外的任何浮点表达式的结果都可能具有与表达式类型不同的浮点类型的范围和精度（请参阅FLT\_EVAL\_METHOD）

notwithstanding the above, intermediate results of any floating-point expression may be calculated as if to infinite range and precision (unless #pragma STDC FP\_CONTRACT is OFF)

尽管如此，任何浮点表达式的中间结果都可以计算为无限范围和精度（除非#pragma STDC FP\_CONTRACT为OFF）

#### Notes

Because the compiler is (usually) unable to analyze the code of an external library to determine whether it does or does not perform I/O or volatile access, third-party library calls also aren't affected by optimization. However, standard library calls may be replaced by other calls, eliminated, or added to the program during optimization. Statically-linked third-party library code may be subject to link-time optimization.

Programs with undefined behavior, e.g. due to access to an array out of bounds, modification of a const object, evaluation order violations, etc, are free from the as-if rule: they often change observable behavior when recompiled with different optimization settings. For example, if a test for signed integer overflow relies on the result of that overflow, e.g. if(n+1 < n) abort();, it is removed entirely by some compilers because signed overflow is undefined behavior and the optimizer is free to assume it never happens and the test is redundant.

Copy elision is an exception from the as-if rule: the compiler may remove calls to move- and copy-constructors and the matching calls to the destructors of temporary objects even if those calls have observable side effects.

#### 笔记

由于编译器（通常）无法分析外部库的代码以确定其是否执行I / O或非易失性访问，因此第三方库调用也不受优化的影响。但是，标准库调用可能会被其他调用所取代，取消或在优化期间添加到程序中。静态链接的第三方库代码可能需要优化链接时间。

未定义行为的程序，例如由于访问数组越界，修改常量对象，评估顺序违规等等都不受as-if规则的限制：它们在使用不同的优化设置重新编译时经常会改变可观察的行为。例如，如果对有符号整数溢出的测试依赖于该溢出的结果，例如，如果（n + 1 <n）abort（），它会被一些编译器完全删除，因为签名溢出是未定义的行为，优化器可以自由地假设它永远不会发生，并且测试是多余的。

复制elision是as-if规则的一个例外：即使这些调用具有可观察的副作用，编译器也可以移除对移动和复制构造函数的调用以及对临时对象的析构函数的匹配调用。

|  |  |
| --- | --- |
| New-expression has another exception from the as-if rule: the compiler may remove calls to the replaceable allocation functions even if a user-defined replacement is provided and has observable side-effects.  New-expression有另一个例外规则：即使提供了用户定义的替换并且具有可观察的副作用，编译器可能会移除对可替换分配函数的调用。 | （C++14之后） |

The count and order of floating-point exceptions can be changed by optimization as long as the state as observed by the next floating-point operation is as if no optimization took place:

只要下一个浮点操作观察到的状态就好像没有进行优化一样，浮点异常的计数和顺序可以通过优化进行更改：

|  |
| --- |
| #pragma STDC FENV\_ACCESS ON  for (i = 0; i < n; i++) x + 1; // x+1 is dead code, but may raise FP exceptions  // (unless the optimizer can prove otherwise). However, executing it n times will  // raise the same exception over and over. So this can be optimized to:  if (0 < n) x + 1; |

12.5.2 拷贝省略（copy elision）

省略拷贝和*移动（C++11之后）*构造函数，产生零拷贝值传递语义。

## 12.6 空基优化

允许空的基(base)子对象的大小为零。

12.6.1 解释

即使类型是空类类型（即没有非静态数据成员的类或结构体），任何对象或成员子对象的大小都必须至少为1，以便能够保证同一类型的不同对象的地址总是不同的。

但是，基类子对象不受如此限制的，且可以从对象布局中完全被优化掉：

|  |
| --- |
| #include <cassert>  struct Base {}； // 空类  struct Derived1 ： Base {  int i；  }；  int main()  {  //任何空类类型的对象的大小至少为1  [assert](http://en.cppreference.com/w/cpp/error/assert)(sizeof(Base) >= 1)；  // 空基优化应用  [assert](http://en.cppreference.com/w/cpp/error/assert)(sizeof(Derived1) == sizeof(int))；  } |

如果其中一个空基类也是第一个非静态数据成员的类型或基类型，则禁止空基优化，因为相同类型的两个基子对象需要在最底层派生对象的表示中具有不同的地址。

一个典型的例子就是，std：：reverse\_iterator（派生于空基类std：：iterator）的简单实现，它将基础iterator（也是派生于std：：iterator）作为它的第1个非静态数据成员。

|  |
| --- |
| #include <cassert>  struct Base {}； // 空类  struct Derived1 ： Base {  int i；  }；  struct Derived2 ： Base {  Base c； // Base, 占据1个字节, 之后填充，以满足对齐要求  int i；  }；  struct Derived3 ： Base {  Derived1 c； // 派生于Base, 占据sizeof(int)个字节  int i；  }；  int main()  {  // 空基优化没有应用，Base成员占据1个字节，随后3个字节的填充，以满足对齐要求  assert(sizeof(Derived2) == 2\*sizeof(int))；    // 空基优化没有应用,Base成员至少占用1个字节，然后加上填充物以满足第一个成员的对齐要求  assert(sizeof(Derived3) == 3\*sizeof(int))；  } |

|  |  |
| --- | --- |
| Empty base optimization is required for StandardLayoutTypes in order to maintain the requirement that the pointer to a standard-layout object, converted using reinterpret\_cast, points to its initial member, which is why one of the requirements for a standard layout type is "has no base classes with non-static data members and has no base classes of the same type as the first non-static data member". | C++11之后 |

StandardLayoutTypes需要空基优化才能保持指向使用reinterpret\_cast转换的标准布局对象的指针指向其初始成员的要求，这就是为什么标准布局类型的其中一项要求是“没有基础”包含非静态数据成员的类，并且没有与第一个非静态数据成员相同类型的基类“。

## 12.7 plmpl

# 13 补充

## 13.1 C++的RTTI观念和用途

自从1993年Bjarne Stroustrup[注1]提出有关C++的RTTI功能之建议，以及C++的异常处理需要RTTI；最近新推出的C++或多或少已提供RTTI。然而，若不小心使用RTTI，可能会导致软件弹性的降低。本文将介绍RTTI的观念和近况，并说明如何善用它。

### 13.1.1什么是RTTI﹖

在C++环境中，头文件含有类定义，亦即包含有关类的表述信息。但是，这些资料只供编译器(compiler)使用，编译完毕后并未留下来，所以在运行时，无法得知对象的类信息，包括类名称、数据成员名称与类型、函数名称与类型等等。例如，两个类Figure和Circle，为继承关系。

若有如下指令﹕

|  |
| --- |
| Figure \*p;  p = new Circle();  Figure &q = \*p; |

在执行时，p指向一个对象，但欲得知此对象的类信息，就有困难了。同样欲得知q引用对象的类信息，也无法得到。RTTI(Run-Time Type Identification)就是要解决这困难，也就是在运行时，想知道指针所指到或参考到的对象类型时，该对象有能力来告诉我们。随着应用场合之不同，所需支持的RTTI范围也不同。最单纯的RTTI包括﹕

* 类识别 - 包括类名称或ID。
* 继承关系 - 支持运行时的「往下变换」(downward casting)，亦即动态变换类型(dynamic casting)。

在对象数据存取上，还需要下述RTTI﹕

* 对象结构(object layout) - 包括属性的类型、名称及其位置（position或offset）。
* 成员函数表(table of functions) - 包括函数的类型、名称、及其参数类型等。

其目的是协助对象的I/O 和持久化(persistence) ，也提供调试讯息等。

若依照Bjarne Stroustrup 之建议[注1]，C++ 还应包括更完整的RTTI﹕

* 能得知类所实例化的各对象。
* 能参考到函数的源代码。
* 能取得类的有关在线说明(on-line documentation)。

其实这些都是C++编译完成时所丢弃的资料，如今只是希望寻找个途径来将之保留到执行期间。然而，要提供完整的RTTI，将会大幅提高C++的复杂度﹗

### 13.1.2 RTTI可能伴随的副作用

RTTI最主要的副作用是﹕程序员可能会利用RTTI来支持其「复选」(multiple-selection)方法，而不使用虚函数(virtual function)方法。

虽然这两种方法皆能达到多态化(polymorphism)，但使用复选方法，常导致违反著名的「开放╱封闭原则」(open/closed principle) [注2]。反之，使用虚函数方法则可合乎这个原则。

Circle和Square皆是由Figure所派生出来的子类，它们各有自己的draw()函数。当C++ 提供了RTTI，就可写个函数如下﹕

void drawing( Figure \*p )

{

if( typeid(\*p).name() == "Circle" )

((Circle\*)p) -> draw();

if( typeid(\*p).name() == "Rectangle" )

((Rectangle\*)p) -> draw();

}

虽然drawing()函数也具有多态性，但它与Figure类体系的结构具有紧密的相关性。当Figure类体系再派生出子类时，drawing()函数的内容必须多加个if指令。因而违反了「开放╱封闭原则」，如下﹕

很显然地，drawing()函数应加以修正。

想一想，如果C++ 并未提供RTTI，则程序员毫无选择必须使用虚函数来支持drawing()函数的多态性。于是程序员将draw()声明为虚函数，并写drawing()如下﹕

void drawing(Figure \*p)

{ p->draw(); }

如此，Figure类体系能随时派生类，而不必修正drawing()函数。亦即，Figure体系有个稳定的接口(interface) ，drawing()，使用这接口，使得drawing()函数也稳定，不会随Figure类体系的扩充而变动。这是封闭的一面。而这稳定的接口并未限制Figure体系的成长，这是开放的一面。因而合乎「开放╱封闭」原则，软件的结构会更具弹性，更易于随环境而不断成长。

### 13.1.3 RTTI的常见的使用场合

一般而言，RTTI的常见使用场合有四﹕异常处理(exceptions handling)、动态转类型(dynamic casting) 、模块集成、以及对象I/O 。

1.异常处理── 大家所熟悉的C++ 新功能﹕异常处理，其需要RTTI，如类名称等。

2.动态转类型── 在类体系(class hierarchy) 中，往下的类型转换需要类继承的RTTI。

3.模块集成── 当某个程序模块里的对象欲跟另一程序模块的对象沟通时，应如何得知对方的身分呢﹖知道其身分资料，才能呼叫其函数。一般的C++ 程序，常见的解决方法是──在源代码中把对方对象之类定义（即存在头文件里）包含进来，在编译时进行连结工作。然而，像目前流行的主从(Client-Server) 架构中，客户端(client)的模块对象，常需与主机端(server)的现成模块对象沟通，它们必须在执行时沟通，但又常无法一再重新编译。于是靠标头文件来提供的类定义资料，无助于执行时的沟通工作，只得依赖RTTI了。

4.对象I/O ── C++ 程序常将其对象存入数据库，未来可再读取之。对象常内含其它小对象，因之在存入数据库时，除了必须知道对象所属的类名称，也必须知道各内含小对象之所属类，才能完整地将对象存进去。储存时，也将这些RTTI资料连同对象内容一起存入数据库中。未来读取对象时，可依据这些RTTI资料来分配内存空间给对象。

### 13.1.4 RTTI从那里来﹖

上述谈到RTTI的用途，以及其副作用。这众多争论，使得RTTI的标准迟迟未呈现出来。也导致各C++ 开发环境提供者，依其环境所需而以各种方式来支持RTTI，且其支持RTTI的范围也所不同。 目前常见的支持方式包括﹕

* 由类库提供RTTI──例如，Microsoft 公司的Visual C++环境。
* 由C++ 编译器(compiler)提供──例如，Borland C++ 4.5 版本。
* 由源代码产生器(code generator)提供──例如Bellvobr系统。
* 由OO数据库的特殊预处理器(preprocessor)提供──例如Poet系统。
* 由程序员自己加上去。

这些方法皆只提供简单的RTTI，其仅为Stroustrup先生所建议RTTI内涵的部分集合而已。相信不久的将来，会由C++ 编译器来提供ANSI标准的RTTI，但何时会订出这标准呢﹖ 没人晓得吧﹗

### 13.1.5 程序员自己提供的RTTI

通常程序员自己可提供简单的RTTI，例如提供类的名称或识别(TypeID)。最常见的方法是﹕为类体系定义些虚函数如Type\_na() 及Isa() 函数等。请先看个例子﹕

class Figure { };

class Rectangle : public Figure { };

class Square : public Rectangle

{

int data;

public:

Square() { data=88; }

void Display() { cout << data << endl; }

};

void main()

{ Figure \*f = new Rectangle();

Square \*s = (Square \*)f;

s -> Display();

}

这时s 指向Rectangle 之对象，而s->Display()呼叫Square::Display() ，将找不到data值。若在执行时能利用RTTI来检查之，就可发出错误讯息。于是，自行加入RTTI功能﹕

class Figure

{ public:

virtual char\* Type\_na() { return "Figure"; }

virtual int Isa(char\* cna) { return !strcmp(cna, "Figure")? 1:0; }

};

class Rectangle:public Figure

{ public:

virtual char\* Type\_na() { return "Rectangle"; }

virtual int Isa(char\* cna) { return !strcmp(cna, "Rectangle")?1 : Figure::Isa(cna); }

static Rectangle\* Dynamic\_cast(Figure\* fg) { return fg -> Isa(Type\_na())?(Rectangle\*)fg : 0; }

};

class Square:public Rectangle

{ int data;

public:

Square() { data=88; }

virtual char\* Type\_na() { return "Square"; }

virtual int Isa(char\* cna) { return !strcmp(cna, "Rectangle")? 1 : Rectangle::Isa(cna); }

static Square\* Dynamic\_cast(Figure \*fg) { return fg->Isa(Type\_na())? (Square\*)fg : 0; }

void Display() { cout << "888" << endl; }

};

虚函数Type\_na() 提供类名称之RTTI，而Isa() 则提供继承之RTTI，用来支持「动态转类型」函数──Dynamic\_cast()。例如﹕

Figure \*f = new Rectangle();

cout << f -> Isa("Square") << endl;

cout << f -> Isa("Figure") << endl;

这些指令可显示出﹕f 所指向之对象并非Square之对象，但是Figure之对象（含子孙对象）。再如﹕

Figure \*f; Square \*s;

f = new Rectangle();

s = Square == Dynamic\_cast(f);

if(!s)

cout << "dynamic\_cast error!!" << endl;

此时，依RTTI来判断出这转类型是不对的。

### 13.1.6 类库提供RTTI

由类库提供RTTI是最常见的，例如Visual C++的MFC 类库内有个CRuntimeClass 类， 其内含简单的RTTI。请看个程序﹕

class Figure:public CObject

{

DECLARE\_DYNAMIC(Figure);

};

class Rectangle : public Figure

{

DECLARE\_DYNAMIC(Rectangle);

};

class Square : public Rectangle

{

DECLARE\_DYNAMIC(Square);

int data;

public:

void Display() { cout << data << endl; }

Square() { data=88; }

};

IMPLEMENT\_DYNAMIC(Figure, CObject);

IMPLEMENT\_DYNAMIC(Rectangle, Figure);

IMPLEMENT\_DYNAMIC(Square, Rectangle);

Visual C++程序依赖这些宏(Macor) 来支持RTTI。现在就看看如何使用CRuntimeClass类吧﹗如下﹕

CRuntimeClass \*r;

Figure \*f = new Rectangle();

r = f -> GetRuntimeClass();

cout << r -> m\_psClassName << endl;

这就在执行时期得到类的名称。Visual C++的类库仅提供些较简单的RTTI──类名称、对象大小及父类等。至于其它常用的RTTI如──数据项的类型及位置(position)等皆未提供。

### 13.1.7 C++编译器提供RTTI

由C++ 语言直接提供RTTI是最方便了，但是因RTTI的范围随应用场合而不同，若C++语言提供所有的RTTI，将会大幅度增加C++ 的复杂度。目前，C++ 语言只提供简单的RTTI，例如Borland C++ 新增typeid()操作数以及dynamic\_cast<T\*>函数样版。请看个程序﹕

|  |
| --- |
| class Figure  { public:  virtual void Display();  };  class Rectangle : public Figure { };  class Square:public Rectangle  { int data;  public:  Square() { data=88; }  void Display() { cout << data << endl; }  }; |

现在看看如何使用typeid()操作数──

Figure \*f = new Square();

const typeinfo ty = typeid(\*f);

cout << ty.name() << endl;

这会告诉您﹕f 指针所指的对象，其类名称是Square。再看看如何使用dynamic\_cast<T\*>函数样版──

Figure \*f; Square \*s;

f = new Rectangle();

s = dynamic\_cast<Sqiare \*>(f);

if(!s)

cout << "dynamic casting error!!" << endl;

在执行时，发现f 是不能转为Square \*类型的。如下指令﹕

Figure \*f; Rectangle \*r;

f = new Square();

r = dynamic\_cast<Rectangle \*>(f);

if(r) r->Display();

这种类型转换是对的。

### 13.1.8 RTTI与虚函数表

在C++程序中，若类含有虚函数，则该类会有个虚函数表（简称VFT）。为了提供RTTI，C++就将在VFT中附加个指针，指向typeinfo对象，这对象包含RTTI信息。

由于该类所实例化的各个对象，皆含有个指针指向VFT表，因之各对象皆可取出typeinfo对象而得到RTTI。例如，

|  |
| --- |
| Figure \*f1 = new Square();  Figure \*f2 = new Square();  const typeinfo ty = typeid(\*f2); |

其中，typeid(\*f2)的动作是：

* 1. 取得f2所指对象。
  2. 从对象取出指向VMF之指针，经由此指针取得VFT表。
  3. 从表中找出指向typeinfo对象之指针，经由此指针取得typeinfo对象。

这typeinfo对象就含有RTTI了。经由f1及f2两指针皆可取得typeinfo对象，所以typeid(\*f2) == typeid(\*f1)。

总结

RTTI是C++的新功能。过去，C++语言来提供RTTI时，大多依赖类库来支持，但各类库使用的方法有所不同，使得程序的可移植性大受影响。然而，目前C++也只提供最简单的RTTI而已，可预见的未来，当大家对RTTI的意见渐趋一致时，C++将会提供更完整的RTTI，包括数据项和成员函数的类型、位置(offset)等资料，使得C++ 程序更井然有序，易于维护。

参考资料

[注1] Stroustrup B., “Run-Time Type Identification for C++”, Usenix C++ C

onference, Portland, 1993.

[注2] Meyer B.,Object-Oriented Software Construction, Prentice Hall, 1988